



C o m m u n i t y   E x p e r i e n c e   D i s t i l l e d

# Learning Google Guice

Utilize dependency injection to develop applications quickly and efficiently using Google Guice

**Hussain Pithawala**

**[PACKT]** open source\*  
PUBLISHING community experience distilled

[www.zhaoziyuan.com](http://www.zhaoziyuan.com)

# Learning Google Guice

Utilize dependency injection to develop applications quickly and efficiently using Google Guice

**Hussain Pithawala**



BIRMINGHAM - MUMBAI

# Learning Google Guice

Copyright © 2013 Packt Publishing

All rights reserved. No part of this book may be reproduced, stored in a retrieval system, or transmitted in any form or by any means, without the prior written permission of the publisher, except in the case of brief quotations embedded in critical articles or reviews.

Every effort has been made in the preparation of this book to ensure the accuracy of the information presented. However, the information contained in this book is sold without warranty, either express or implied. Neither the author, nor Packt Publishing, and its dealers and distributors will be held liable for any damages caused or alleged to be caused directly or indirectly by this book.

Packt Publishing has endeavored to provide trademark information about all of the companies and products mentioned in this book by the appropriate use of capitals. However, Packt Publishing cannot guarantee the accuracy of this information.

First published: September 2013

Production Reference: 1170913

Published by Packt Publishing Ltd.  
Livery Place  
35 Livery Street  
Birmingham B3 2PB, UK.

ISBN 978-1-78328-189-3

[www.packtpub.com](http://www.packtpub.com)

Cover Image by Abhishek Pandey ([abhishek.pandey1210@gmail.com](mailto:abhishek.pandey1210@gmail.com))

# Credits

**Author**

Hussain Pithawala

**Project Coordinator**

Akash Poojari

**Reviewers**

Dmitry Spikhalskiy

Jari Timonen

**Proofreader**

Clyde Jenkins

**Acquisition Editor**

James Jones

**Indexer**

Tejal Soni

**Commissioning Editor**

Manasi Pandire

**Graphics**

Ronak Dhruv

Abhinash Sahu

**Technical Editors**

Dipika Gaonkar

Mrunmayee Patil

Sonali Vernekar

**Production Coordinator**

Nitesh Thakur

**Cover Work**

Nitesh Thakur

# About the Author

**Hussain Pithawala** is a polyglot programmer, a technical blogger, open source enthusiast, and a certified scrum master.

At present he is working as technical lead at Synerzip Softech India Pvt. Ltd, a leading dual-shore agile software product development company located in Pune, India.

For the past seven years, he has been working in I.T. industry in various domains such as, office productivity suite, telecom provisioning, travel, and E-commerce.

He holds expertise in software tooling, BPMN-based workflow engines, scripting languages, enterprise application development, mobile application development, and data analytics.

LinkedIn Profile: <http://in.linkedin.com/in/hussainpithawala>

Twitter: <https://twitter.com/hussainpw>

Blog: <http://hussainpithawala.wordpress.com>

# Acknowledgments

I express my gratitude to Bob Lee, founder of Google Guice for creating such a masterpiece, Guice and to Google for fostering open source movement.

Many thanks to the Packt Publishing team for helping me to author this book. I had a tremendous and enjoyable experience working with all of you.

At the same time, technical reviews from Jari Timonen and Dmitry Spikhalskiy were critically helpful in crafting the book to its final shape. I sincerely thank you both for your expert advice and comments.

At last, thanks to my colleagues at Synerzip; Prafulla Kelkar, Alok Guha, and Atul Moglewar for reviewing early drafts and helping me to gauge a reader's perspective.

# About the Reviewers

**Dmitry Spikhalskiy** is currently holding a position of technical lead and architect in a mobile social banking startup called Instabank.

Previously, as a senior developer, took part in developing the Mind Labs' platform, infrastructure and benchmarks for highload video conference and streaming service, which got "The biggest online-training in the world" Guinness World Record with more than 12,000 participants.

He graduated from Moscow State University with an M.Sc. degree in Computer Science, where he first got interested in parallel data processing, highload systems, and databases.

---

I want to thank my mother, girlfriend, and all my friends and colleagues for help and patience for reviewing this book.

---

**Jari Timonen** is an experienced software enthusiast with over 10 years of experience in software industry. His experience includes successful team leadership combined with understanding complex business domains and delivering them into practice. He has been building enterprise architectures, designing software, and programming. While starting his career in the finance industry, he currently works as Solution Architect in a telecommunications company. He practices pair programming and is keen on studying new technologies. He has also reviewed *Spring Data*, Packt Publishing. When he is not building software, he is spending time with his family, fishing, or flying his radio-controlled model helicopter.

He currently holds certifications in Sun Certified Programmer for the Java 2 Platform, Standard Edition 5 (SCJP), Sun Certified Developer for the Java 2 Platform (SCJD) and Oracle Certified Master, and Java EE 5 Enterprise Architect (OCMJEA).

# www.PacktPub.com

## Support files, eBooks, discount offers, and more

You might want to visit [www.PacktPub.com](http://www.PacktPub.com) for support files and downloads related to your book.

Did you know that Packt offers eBook versions of every book published, with PDF and ePub files available? You can upgrade to the eBook version at [www.PacktPub.com](http://www.PacktPub.com) and as a print book customer, you are entitled to a discount on the eBook copy. Get in touch with us at [service@packtpub.com](mailto:service@packtpub.com) for more details.

At [www.PacktPub.com](http://www.PacktPub.com), you can also read a collection of free technical articles, sign up for a range of free newsletters and receive exclusive discounts and offers on Packt books and eBooks.



<http://PacktLib.PacktPub.com>

Do you need instant solutions to your IT questions? PacktLib is Packt's online digital book library. Here, you can access, read and search across Packt's entire library of books.

## Why Subscribe?

- Fully searchable across every book published by Packt
- Copy and paste, print and bookmark content
- On demand and accessible via web browser

## Free Access for Packt account holders

If you have an account with Packt at [www.PacktPub.com](http://www.PacktPub.com), you can use this to access PacktLib today and view nine entirely free books. Simply use your login credentials for immediate access.





*To Tasneem my wife, Mohammad my son, Shirin my mom, Fazle Abid my dad  
with all my love for your encouragement, support, patience, and love.*



# Table of Contents

<b>Preface</b>	<b>1</b>
<b>Chapter 1: Getting Started with Guice</b>	<b>7</b>
<b>Wiring dependencies</b>	<b>7</b>
Resolving dependencies directly	7
Inverting the dependencies	8
Understanding Inversion of Control	8
Inverting the control using the Dependency injection	9
<b>Comparing two approaches</b>	<b>10</b>
Initializing dependencies directly	10
Refactoring to use Guice	12
<b>Building and Running the examples</b>	<b>14</b>
Compiling the sample code	15
Running Unit tests	15
Running the main application	15
<b>Summary</b>	<b>15</b>
<b>Chapter 2: Exploring Google Guice</b>	<b>17</b>
<b>Meet the injector</b>	<b>17</b>
Various kinds of injections	18
Field injection	18
Method injection	18
Constructor injection	18
<b>Configuring a module to write bindings</b>	<b>19</b>
Bindings	20
Linked bindings	20
Instance bindings	21
Untargeted bindings	21
Constructor bindings	21
Binding annotations	23

Binding constants	24
Binding properties	25
Just in time bindings	26
By default constructors	26
@Inject annotated constructors	26
Interfaces annotated with @ImplementedBy	26
<b>Static injection</b>	<b>27</b>
<b>Summary</b>	<b>27</b>
<b>Chapter 3: Diving Deeper in Guice</b>	<b>29</b>
<b>Going the extra mile with Providers</b>	<b>29</b>
Need for a Provider	29
Working of a Provider	30
Rolling your own Provider	30
Injecting dependencies in Providers	31
Advantages of a Provider	32
@Provides	32
CheckedProviders	33
@CheckedProvides	35
<b>AssistedInject</b>	<b>35</b>
<b>Binding collections</b>	<b>37</b>
Using TypeLiteral	38
Using MultiBinder	38
<b>Scoping</b>	<b>39</b>
Singletons with Guice	39
Eager Singletons	40
Custom Scopes	41
Motivation to write a custom scope	41
Modifying our Provider	42
Defining our own scope	43
Changes in Module	43
Observations	45
<b>Summary</b>	<b>45</b>
<b>Chapter 4: Guice in Web Development</b>	<b>47</b>
<b>Structure of flightsweb application</b>	<b>47</b>
<b>Using GuiceFilter</b>	<b>49</b>
<b>Rolling out our ServletContextListener interface</b>	<b>49</b>
<b>ServletModule – the entry point for configurations</b>	<b>51</b>
<b>Binding language</b>	<b>51</b>
Mapping servlets	52
Mapping filters	53

---

<b>Web scopes</b>	<b>54</b>
@RequestScoped	55
@SessionScoped	55
@RequestParameters	56
Exercising caution while scoping	57
<b>Summary</b>	<b>58</b>
<b>Chapter 5: Integrating Guice with Struts 2</b>	<b>59</b>
<b>Introducing Struts 2</b>	<b>60</b>
Guice 3 and Struts 2 integration	62
Struts2GuicePluginModule	62
Struts2Factory	63
FlightSearch application with Struts 2	63
FlightServletContextListener	63
Index page	63
FlightSearchAction	64
Response page	65
<b>Summary</b>	<b>65</b>
<b>Chapter 6: Integrating Guice with JPA 2</b>	<b>67</b>
<b>Introduction to JPA 2 and Hibernate 3</b>	<b>68</b>
Persistence Unit	68
PersistenceContext	68
EntityManagerFactory	69
EntityManager	69
Session Strategies	69
<b>Guice 3 with JPA 2</b>	<b>69</b>
PersistService	70
UnitOfWork	70
JpaPersistService	70
PersistFilter	70
@Transactional	71
JpaLocalTxnInterceptor	71
JpaPersistModule	72
<b>FlightSearch application with JPA 2 and Hibernate 3</b>	<b>73</b>
Persistence.xml	73
Client	73
SearchResponse	74
FlightEngineModule	75
FlightJPASupplier	75
FlightServletContextListener	76
<b>Summary</b>	<b>77</b>

---

<b>Chapter 7: Developing Plugins and Extensions using Guice</b>	<b>79</b>
<b>Developing a plugin</b>	<b>79</b>
<b>Guice SPI (Service Provider Interface)</b>	<b>80</b>
Quick introduction to Visitor Design pattern	81
Elements SPI	83
Implementing Elements SPI	84
Uses of Elements SPI	85
Extensions SPI	86
<b>Exploring a custom extension, guice-multibinder</b>	<b>89</b>
Important classes and interfaces in guice-multibinder	90
Multibinder in action	91
<b>Summary</b>	<b>93</b>
<b>Chapter 8: AOP with Guice</b>	<b>95</b>
<b>What is AOP?</b>	<b>95</b>
<b>How AOP works?</b>	<b>95</b>
<b>How Guice supports AOP?</b>	<b>96</b>
Implementing a LoggingInterceptor	96
Understanding Matchers	99
Limitations	101
<b>Concerns related to performance</b>	<b>102</b>
<b>Summary</b>	<b>102</b>
<b>Appendix: Prerequisites</b>	<b>103</b>
<b>Getting the sample code</b>	<b>103</b>
Installing Git	103
Cloning the repository	104
<b>JDK</b>	<b>104</b>
<b>Installing Maven</b>	<b>104</b>
Mac OSX and Linux	104
Microsoft Windows	105
<b>Installing Eclipse</b>	<b>105</b>
<b>Installing the m2eclipse plugin</b>	<b>105</b>
<b>Installing Apache Tomcat</b>	<b>106</b>
<b>Installing MySQL</b>	<b>106</b>
Microsoft Windows	106
Mac OSX	107
Linux	107
<b>Importing MySQL data</b>	<b>107</b>
<b>Index</b>	<b>109</b>

---

# Preface

Guice appeared amidst various dependency injection frameworks in March 2007. Handcrafted in Google, Guice offered altogether new experience of developing applications in Java without having to bother about developing a Factory or calling new keywords to create instances altogether. After six years, three major releases and a number of extensions around the core, Guice is thriving. Today, Guice 3.0 stands as a complete implementation of JSR-330 Injector. Bob Lee, founder of Guice leading the JSR-330 gives a great impression that philosophy of dependency injection is making great strides towards improving the way we program in Java. That itself is a big reason to get ones hands wet with Guice.

My first interaction with Guice happened in fall of 2010, when I was developing a Maven plugin for code scaffolding in Java. As I had a large code base to generate the source code itself, wiring dependencies using factories or injecting them through a constructor was out of question. I investigated various frameworks available for dependency injection. While most of the frameworks did focus on dependency injection yet none of them had such variety of rich APIs to bind dependencies to their implementations in various possible ways. This made Guice my first choice.

This book is written to share my learning and experiences, which I had while using Guice. With this book, the readers have with them detailed hands-on treatment to Guice along with a number of extensions, which surround it. The book is based on latest version of Guice, Guice 3.0. It focuses on core functionality, latest features offered and extensions surrounding the core functionality (to solve specific problems or to provide exclusive features).



## What this book covers

*Chapter 1, Getting Started with Guice*, introduces Guice as an alternative to writing dependency injection code using constructors or factories. The focus is on simplicity and separation of concerns, which Guice brings in with dependency injection. Along with this, Maven is introduced to build and run the sample applications.

*Chapter 2, Exploring Google Guice*, lays down a sound footing in various areas of the dependency injection. We start our discussion with Injector, which prepares the object graph and injects the dependencies. Discussion continues with various types of bindings, Guice provides to bind a reference to its implementation. At the end of the chapter, we visit a case for injecting static dependencies.

*Chapter 3, Diving Deeper in Guice*, focuses on advanced concepts in Guice. Beyond explicit bindings we investigate the application of Providers, Scoping, Binding collections, and Configuration Management. Various Guice extensions such as guice-throwingproviders, guice-assistedinject, and guice-multibindings are discussed. A solution to handle installation of modules is discussed in later part.

*Chapter 4, Guice in Web Development*, helps us to learn about developing a web application in java using Servlets and JSPs using Guice. We discuss the programmatic approach to configure routing to servlets and JSPs and avoiding declarative approach altogether in web.xml. We discuss guice-servlet an important extension, which provides important artifacts like GuiceServletContextListener, ServletModule, and GuiceFilter. We learn how to scope various instances to Request, Session, and Singleton.

*Chapter 5, Integrating Guice with Struts 2*, demonstrates integration of Guice with Struts 2. We discuss guice-struts the extension, which provides various artifacts such as Struts2GuicePluginModule and Struts2Factory for easy integration. The web application developed is redeveloped with Guice and Struts2.

*Chapter 6, Integrating Guice with JPA 2*, illustrates integration of Guice with JPA 2 and Hibernate 3 in a web and standalone application. We continue with our Struts 2 based application from the previous chapter and make it driven by a JPA, Hibernate-backed database. In this process we discuss the extension guice-persist and various artifacts it provides such as, JpaPersistModule and PersistFilter. Transaction handling is discussed using JpaTxnInterceptor.

*Chapter 7, Developing Plugins and Extensions using Guice*, focuses on SPI, Service Provider Interface. Important classes and interfaces, which are useful for developing extensions and plugins around Guice, are discussed. A custom LoggingInterceptor is developed to understand various functionalities of SPI. guice-multibinder, an extension for binding collections is analyzed.

*Chapter 8, AOP with Guice*, provides a brief overview of what AOP is, and how Guice adds to its own style of AOP with method interceptors by providing an implementation of aopalliance. Various artifacts such as Matchers and MethodInterceptor are discussed. A lot of examples are covered detailing both the artifacts.

*Appendix, Prerequisites*, provides tips regarding essential tools involved in running and building the samples. Installation of Maven, Eclipse, Tomcat, and MySQL server is illustrated. Importing MySQL data dump for running the samples of chapter 6, 7, and 8 is also detailed.

## What you need for this book

You need to have JDK 1.5 installed on your machine. For running the samples from the various chapters Maven, the ubiquitous build and project management tool is required. Readers are advised to refer to the Appendix for prerequisites while running the samples.

## Who this book is for

This book is for anyone hands on with Java (JDK 5 and above) and curious to implement Guice for dependency injection. Being hands on book, programmers struggling with Guice would also find it as a handy reference.

## Conventions

In this book, you will find a number of styles of text that distinguish between different kinds of information. Here are some examples of these styles, and an explanation of their meaning.

Code words in text are shown as follows: "We can include other contexts through the use of the `include` directive."

A block of code is set as follows:

```
new ServletModule() {  
    @Override  
    protected void configureServlets() {  
        install(new MainModule());  
        serve("/response").with(FlightServlet.class);  
        serve("/").with(IndexServlet.class);  
    }  
}
```


When we wish to draw your attention to a particular part of a code block, the relevant lines or items are set in bold:


```
SearchRQ create(  
    @Assisted("depLoc") String depLoc,  
    @Assisted("arrivLoc") String arrivLoc,  
    Date flightDate);
```

Any command-line input or output is written as follows:

```
# mvn exec:java -Dexec.mainClass="org.packt.client.Client"
```

**New terms** and **important words** are shown in bold. Words that you see on the screen, in menus or dialog boxes for example, appear in the text like this: "clicking the **Next** button moves you to the next screen".

[  Warnings or important notes appear in a box like this. ]

[  Tips and tricks appear like this. ]

## Reader feedback

Feedback from our readers is always welcome. Let us know what you think about this book – what you liked or may have disliked. Reader feedback is important for us to develop titles that you really get the most out of.

To send us general feedback, simply send an e-mail to [feedback@packtpub.com](mailto:feedback@packtpub.com), and mention the book title through the subject of your message.

If there is a topic that you have expertise in and you are interested in either writing or contributing to a book, see our author guide on [www.packtpub.com/authors](http://www.packtpub.com/authors).

## Customer support

Now that you are the proud owner of a Packt book, we have a number of things to help you to get the most from your purchase.

## Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you. The sample code for this book is also available as a Github repository at [https://bitbucket.org/hussain-pithawala/begin\\_guice](https://bitbucket.org/hussain-pithawala/begin_guice). Readers are encouraged to fork the repository and tweak it for themselves.

## Errata

Although we have taken every care to ensure the accuracy of our content, mistakes do happen. If you find a mistake in one of our books – maybe a mistake in the text or the code – we would be grateful if you would report this to us. By doing so, you can save other readers from frustration and help us improve subsequent versions of this book. If you find any errata, please report them by visiting <http://www.packtpub.com/support>, selecting your book, clicking on the **errata submission form** link, and entering the details of your errata. Once your errata are verified, your submission will be accepted and the errata will be uploaded to our website, or added to any list of existing errata, under the Errata section of that title.

## Piracy

Piracy of copyright material on the Internet is an ongoing problem across all media. At Packt, we take the protection of our copyright and licenses very seriously. If you come across any illegal copies of our works, in any form, on the Internet, please provide us with the location address or website name immediately so that we can pursue a remedy.

Please contact us at [copyright@packtpub.com](mailto:copyright@packtpub.com) with a link to the suspected pirated material.

We appreciate your help in protecting our authors, and our ability to bring you valuable content.

## Questions

You can contact us at [questions@packtpub.com](mailto:questions@packtpub.com) if you are having a problem with any aspect of the book, and we will do our best to address it.



# 1

## Getting Started with Guice

We begin with discussing what are dependencies and how they are resolved directly in a conventional application development. Later on we will discuss the demerits of direct resolution of dependencies, and how to address it using the dependency inversion principle. Following this we move on to discuss Inversion of Control, which is further discussed in one of its most popular ways, Dependency Injection (DI). Later, we will discuss how dependency injection works better when delegated to **Guice**.

### Wiring dependencies

An object-oriented approach to application development requires breaking up of application logic into various classes. A complex operation could be simply viewed as an abstraction in a high-level class, which is carried out by various low-level classes. The association of low-level class references in a high-level class is termed as a **dependency**. Dependencies are resolved by writing application glue code. In its most simple form it could be, direct instantiation of low-level classes instances against their references in high-level classes.

### Resolving dependencies directly

Resolving dependencies directly works well, if the application logic is trivial. With bigger applications, where application logic itself is complex, mingling it up with application glue code could be troublesome. Mixing up of application logic with application glue code defeats the basic purpose of separation of concerns. Another demerit of direct dependency resolution is that the high-level classes are simply dependent on the low-level classes, which reduces the flexibility on both the ends. The reusability gets limited as coupling between both increases.

## Inverting the dependencies

In order to achieve maximum reuse, classes are developed in separate packages as per the functionality, which we generally refer as high-and low-level classes. For example, a class encapsulating the business logic could be termed as the high-level class while a class managing the database operation could be termed as a low-level class. Interfaces, which define behavior or services required by the high-level classes, are kept in the high-level class packages. Implementation of these interfaces by the low-level classes inverts the dependency, which makes sure that now the low-level classes are dependent on high-level interfaces.



### Dependency Inversion Principle

[http://en.wikipedia.org/wiki/Dependency\\_inversion\\_principle](http://en.wikipedia.org/wiki/Dependency_inversion_principle) details out how inverting the dependency lays down the footing for loose coupling between the various layers.

## Understanding Inversion of Control

Conventionally, the low-level objects are statically assigned to the references in the high-level classes. After inverting the dependencies, we refer to the interfaces, which are being implemented by the low-level classes and are present as the association in the high-level classes. Inversion of Control assigns instances to these dependent references using an assembler, which prepares an object graph to determine what implementation should be assigned to an interface reference.

An object graph is prepared by the assembler to determine what implementation is to be assigned to a particular interface. For example, a low-level class which implements an interface from the high-level package needs to be instantiated for being assigned to a reference in the high-level class. This architectural principle harnesses dependency inversion principle to get rid of application glue code written with application logic.



Inversion of Control is a programming technique in which object coupling is bound at run time by an assembler object and is typically not known at compile time using static analysis. For various ways in which Inversion of Control could be achieved, visit the wiki page for Inversion of Control:

[http://en.wikipedia.org/wiki/Inversion\\_of\\_control](http://en.wikipedia.org/wiki/Inversion_of_control)

For a detailed treatment over the subject, readers are advised to visit Martin Fowlers blog on Inversion of Control:

<http://martinfowler.com/bliki/InversionOfControl.html>

## Inverting the control using the Dependency injection

Dependency injection is a way to allow the removal of hard coded dependencies and make it possible to change them whether at compile time or run time.



Dependency injection is an architectural principal and a design pattern. It is one of the ways Inversion of Control is achieved in practice, and is a guiding principle behind frameworks like Spring and Guice. Dependency injection is a subject matter of many books and articles. Readers who are more interested in detailed treatment of the subject are suggested to visit: [http://en.wikipedia.org/wiki/Dependency\\_injection](http://en.wikipedia.org/wiki/Dependency_injection)  
A few of the excerpts from the wiki page are copied here.

Dependency injection involves at least three components:

- A dependent consumer
- A declaration of component's dependencies, defined as interface contracts
- An injector (sometimes referred as provider or container) which creates instances of classes that implement a given dependency interface on request

The dependent object describes what software component it depends on to do its work. The injector or assembler decides what concrete classes satisfy the requirements of the dependent objects and provides them the dependent.

Manually implementing dependency injection poses the following challenges:

- A mechanism to read the configuration either programmatically through APIs or declaratively using XML
- Manually preparing the assembler or injector to inject the dependencies
- Manually preparing the object graph to accomplish contextualized lookups

Considering these challenges, usage of dependency injection frameworks, such as Guice, is advisable against trying to implement the dependency injection manually for an application.

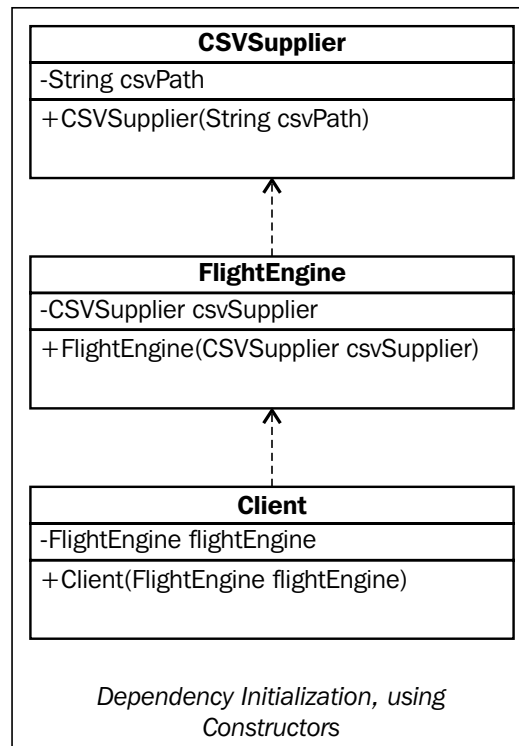


## Comparing two approaches

We will now have a look at two applications with the same application logic. One is developed using direct dependency initialization and other using dependency injection with Guice. Though trivial, these applications help us better understand what we have theoretically discussed so far.

### Initializing dependencies directly

We will first try to develop the application wherein we try to initialize dependencies and assign them manually. Let's consider a fictitious application for searching flights, where we try to prepare and inject dependencies manually via a constructor. A `FlightEngine` is a class, which processes the request via a value object `SearchRequest` and returns the response via a value object `SearchResponse`. `FlightEngine` has a dependency on `CSVSupplier`. `CSVSupplier` is again a class to prepare the value objects after parsing the `.csv` files present in a directory provided. A `Client` class is the invoker of `processRequest` (an API in `FlightEngine.java` to return a `Set<SearchResponse>` for a `SearchRequest` parameter). So, `Client` has a dependency on `FlightEngine`.





### Downloading the example code

You can download the example code files for all Packt books you have purchased from your account at <http://www.packtpub.com>. If you purchased this book elsewhere, you can visit <http://www.packtpub.com/support> and register to have the files e-mailed directly to you.

Constructor from `CSVSupplier`, which takes path to CSV directory as a parameter:

```
public CSVSupplier(String csvPath) {
    this.csvPath = csvPath;
}
```

Constructor of `FlightEngine`, which takes a reference to `CSVSupplier` as a parameter:

```
public FlightEngine(CSVSupplier csvSupplier) {
    this.csvSupplier = csvSupplier;
}
```

Constructor of `Client`, which takes a reference to `FlightEngine` as a constraint:

```
public Client(FlightEngine flightEngine){
    this.flightEngine = flightEngine;
}
```

Client initializing code further illustrates dependency initialization:

```
CSVSupplier csvSupplier = new CSVSupplier("./flightCSV/");
FlightEngine flightEngine = new FlightEngine(csvSupplier);
Client client = new Client(flightEngine);
client.makeRequest();
```

While initializing the `Client` class, the onus of initializing and setting in the dependency falls on to the dependency initialization implementation. The object look-up graph for dependency could grow large and could soon become unmanageable. Such a way of initializing and resolving dependencies is very difficult to maintain, because the developer needs to take care of resolving at various places, and possibly no single place to configure things also adds to confusion. The loose coupling and separation of application glue code from application logic apparently are not achievable with this approach.

You can find sample application within the code file in `chapter1/flights`. For a better understanding, we could import this as an Eclipse project and build it with maven.

## Refactoring to use Guice

Let's return to DI principles and begin refactoring. A DI container should provide:

- An injector, which prepares the object graph to inject the dependencies. It should take care as what implementation should be injected against a particular interface reference. Guice provides `com.google.inject.Injector` to address this problem. It builds the object graph, tracks the dependencies for each type, and forms what is core of a dependency injection framework.
- A declaration of component dependencies. A mechanism is required to provide configuration for the dependencies, which could serve as the basis for binding. Assuming that two implementations of an interface exist. In such a case, a binding mechanism is required as what implementation should be injected to a particular interface reference. Guice provides `com.google.inject.AbstractModule` to provide a readable configuration. We need to simply extend this abstract class and provide configurations in it. The `protected abstract void configure()` method is overridden, which is a place to write configurations referred to as bindings.

A dependent consumer. Guice provides `@Inject` annotation to indicate that a consumer is dependent on a particular dependency. Injector takes care of initializing this dependency using the object graph.

Let's have a look at the code now:

```
class CSVSupplierModule extends AbstractModule{
    @Override
    public void configure() {
        bind(String.class). annotatedWith(Names.named("csvPath")).
        toInstance("./flightCSV/");
        bind(FlightSupplier.class).to(CSVSupplier.class);
    }
}
```

Here, the `configure` method provides us a place to write bindings. Next, look at the injection part in `CSVSupplier`.

```
@Inject @Named("csvPath")
private String csvPath;
```

Let's examine the injection as well as the binding one-by-one. Here in the injection part, we ask to inject a dependency using the `@Inject` annotation of Guice. Next, more explicitly, we ask to inject a dependency only if it is annotated with `@Named` (which is again a Guice-provided annotation). The `@Named` annotation helps to identify different dependencies of same type. For example, there could be many `String` references to be injected. For differentiating we used the `@Named` annotation with a string identifier. In this case, it is `csvPath`. Now, we'll look at the Binding. We invoke the `bind(String.class)` method in order to wire a dependency which is annotated with the `@Named` annotation, and distinguished by identifier `csvPath`. To this dependency, we wire an instance of `String` type `./flightCSV`.

Now, we would bootstrap the injection; all we have to do now is to create an injector and prepare the instance of the `Client`. Following is the excerpt from the `main()` method in `Client`.

```
import com.google.inject.*;

Injector injector = Guice.createInjector(new CSVSupplierModule());
Client client = injector.getInstance(Client.class);
client.makeRequest();
```

We prepare the instance of injector, which is used to prepare instances of our classes. `Guice.createInjector(...)` accepts variable number of arguments of type `Module`. At the next statement we invoke `getInstance(Client.class)` to fetch the instance of `Client`. `Client` instance is now available to process requests.

Wait; how come the dependencies of `FlightSupplier` in `FlightEngine`, and `FlightEngine` in `Client` were injected? Just have a quick look at the `FlightEngine` and `Client` classes. Here, we have declared that we need injections to these references:

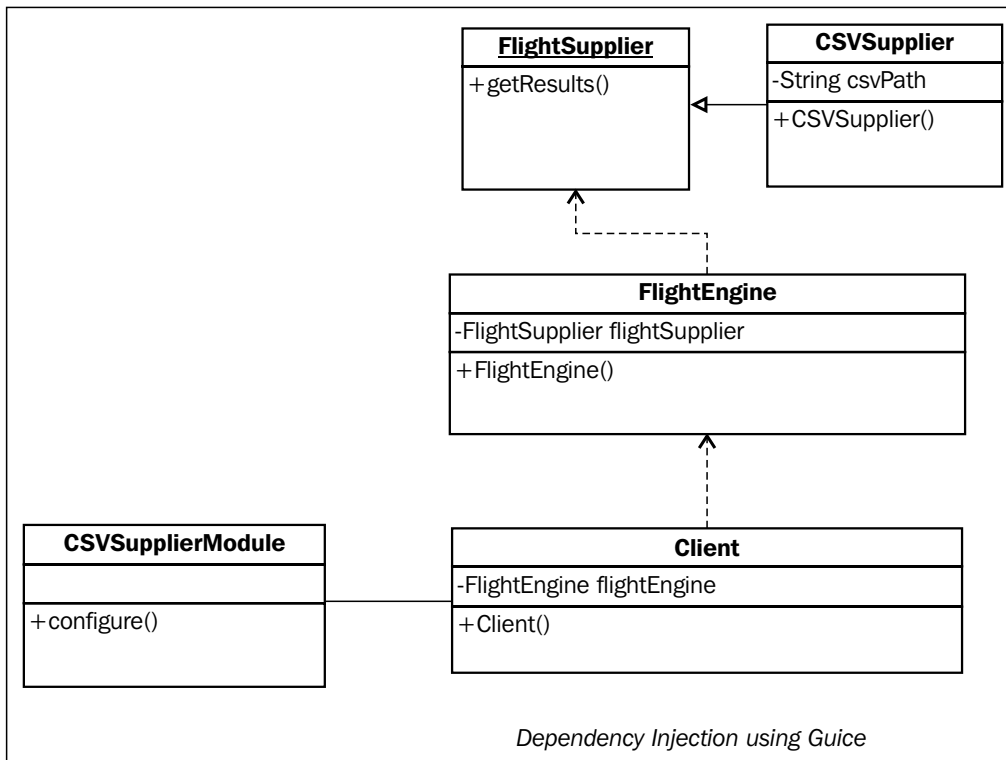
```
@Inject
private FlightEngine flightEngine;

@Inject
private FlightSupplier flightSupplier;
```

Guice, while preparing the dependency graph, resolves dependencies implicitly, if they are not defined explicitly. `FlightEngine` could be directly injected, because it has a default constructor. For `FlightSupplier`, we bind the implementation to `CSVSupplier` with the following statement:

```
bind(FlightSupplier.class).to(CSVSupplier.class);
```

`CSVSupplier` is also instantiated directly, because it also has a default constructor available. This example covers two cases one of implicit binding where the type is statically defined and a default constructor is available for the class and other, where for a particular interface reference we need to mention an implementation explicitly. This gives us the facility to bind different instances against a reference.



## Building and Running the examples

The sample applications are organized as a maven project, which helps us to get rid of routine tasks of downloading jars, manually compiling the code, packaging the application, and running it. If not aware about Maven, refer to *Appendix, Prerequisites* as how to download and install it. Also, it is suggested to import sample application (which are maven projects) into Eclipse IDE for easier reference.

Various directories in a chapter represent one complete application developed to understand a particular scenario. For example, `chapter_1/flights` is the application with vanilla dependency initialization. `chapter_1/flights_with_guice` is the application retrofitted to use Guice's dependency injection features.

## Compiling the sample code

Unzip the code sample, change working directory to `chapter_1/flights_with_guice`, and execute the following command:

```
Shell> mvn clean compile
```

This would execute the various phases of project lifecycle like, cleaning up the previous build (initially just removing and recreating target directory), downloading the various jar files to compile the application. It is suggested that the reader makes frequent changes in source code, as we progress through the each chapter and build, deploy the example as a part of learning.

## Running Unit tests

For running unit tests, execute the following command:

```
Shell> mvn test
```

## Running the main application

For running the code from the class having main method execute:

```
Shell> mvn exec:java -Dexec.mainClass="org.packt.client.Client"
```

## Summary

We briefly discussed the need of dependency injection, and how direct dependency initialization could create problems. Later, we brought Guice to rescue us from this problem. We saw how to configure simple bindings, inject constants, and wire dependencies.

We also saw the two sample applications, one with vanilla dependency initialization and other with Google Guice. We build and ran the sample code using maven. By now, we should be ready to start exploring Google Guice to solve more challenges, which we will face in developing the application further.

In the next chapter we will discuss various ways of dependency injection using Guice, different types of bindings, and other commonly used features.



# 2

## Exploring Google Guice

This chapter lays down a solid foundation in various areas of **dependency injection**. We start our discussion with **injector**, which prepares the object graph and injects the dependencies. Later on we will discuss various types of bindings. These include lots of possibilities, which we could encounter in a practical application development scenario. We also discuss how implicit binding works, when there is no explicit binding provided. At the end of the chapter, we will visit a case for injecting static dependencies.

### Meet the injector

The dependency injection starts with separating behavior from dependency resolution. Dependency injection calls for the dependencies to be passed in rather than calling the dependencies from factories or invoking constructors directly.

For accomplishing this, an injector comes into picture. An injector object basically picks up all declared configuration in modules and prepares a graph of objects. Once request for an instance is made, it does two things:

- Resolves the dependencies using configurations, which are called bindings. Bindings could be explicit or implicit.
- Wires the object graph to instantiate the requested object.

Injector could be initialized using any of the four available static APIs in the `Injector` class.

The preferred API is the one, which we use. It basically accepts the variable number of arguments of `com.google.inject.Module` type. Here, we could pass all the instances of various modules, which have binding configurations.

```
Injector injector = Guice.createInjector(...);
```



Injector then could be used to create the instances of the objects, which we need in our application logic. The instances which are created using injector are termed as the Guice managed objects. For all the Guice managed objects, which are having various dependencies annotated with `@Inject`, dependency injection is done by injector. For example, following is a way to prepare an instance of `FlightEngine`.

```
FlightEngine flightEngine = injector.getInstance(FlightEngine.class);
```

## Various kinds of injections

There are three ways of injecting dependencies:

- Field injection
- Method injection
- Constructor injection

### Field injection

The kind of injection we have used so far is basically the field injection. A field if annotated with `@Inject` annotation gets dependency injected in it. The member variables often have accessibility as `private`, which makes them non-testable, so it is not a preferred way.

```
@Inject
private FlightSupplier flightSupplier;
```

### Method injection

In case we need to have our fields properly testable, we usually declare them as `private`, along with getter and setter APIs. We could inject the dependency by applying the `@Inject` annotation over the setter API, which is in fact a suggested way.

```
@Inject
public void setFlightSupplier (FlightSupplier flightSupplier) {
    this.flightSupplier = flightSupplier;
}
```

This way of injecting dependencies is termed as method injection.

### Constructor injection

Constructor injection is a way to combine instantiation of objects with dependency injection. Assume a case where in the constructor for `FlightEngine` is changed and `FlightSupplier` needs to be supplied as an argument to the constructor.

In such a case, all we need to do is to annotate the `FlightEngine` constructor with `@Inject` annotation. In this way the injector while instantiation of the `FlightEngine` instance, injects the dependency directly.

```
@Inject
public FlightEngine(CSVSupplier flightSupplier) {
    this.flightSupplier = flightSupplier;
}
```

## Configuring a module to write bindings

There are two ways by which we can configure the bindings:

- Firstly by extending the abstract class `com.google.inject.AbstractModule`

```
@Override
protected void configure() {
}
```

- Secondly by implementing the interface `com.google.inject.Module`

```
@Override
public void configure(Binder binder) {
}
```



It is suggested that modules must be extended using `AbstractModule`. It provides a more readable configuration and also avoids repetition of invoking methods on binder.

We have simply followed one module per class strategy. Also, for the sake of illustration, we have developed all the modules as the method local classes in the `main()` method of `Client.class`. In the next chapter we will discuss more about structuring modules in a better way.

Developing modules as method local classes is not a suggested approach. It makes an application difficult to maintain. We have followed this approach for illustration purposes only. In the next chapter modules are developed as independent classes, in a separate package.

Also, if an application has limited number of configurations, they could be combined in a single module. For such applications, single module per package or single module per application would be an appropriate strategy. It is left to the discretion of reader to pick up an appropriate strategy depending upon various factors, such as multiple environments or any need to mock backend.

## Bindings

Binding is perhaps the most important part of the dependency injection. Binding defines how to wire an instance of a particular object to an implementation. Let us visit various cases of bindings one by one.

### Linked bindings

Linked binding helps us map a type to its implementation. This could be of the following type:

- An interface to its implementation
- A super class to a sub class.

Let us have a look at the `FlightSupplier` interface:


```
public interface FlightSupplier {  
    Set<SearchResponse> getResults();  
}
```

Following binding declaration binds an interface against its implementation. Here, `FlightSupplier` is bound to the `CSVSupplier` instance:

```
bind(FlightSupplier.class).to(CSVSupplier.class);
```

In case binding to a different implementation of `FlightSupplier`, `XMLSupplier` is required, we need to simply change the binding:

```
bind(FlightSupplier.class).to(XMLSupplier.class);
```

 What if you try to provide both the declarations for both the implementations in `FlightEngineModule`? This would actually result in a configuration error, which says that a binding to `FlightSupplier` is already present in `FlightEngineModule`.

An important thing to be noted is that linked binding could even be chained. For instance, if we want `CSVSupplier` to be wired to a particular class, which extends the `CSVSupplier`, then we would do something like this:

```
bind(FlightSupplier.class).to(CSVSupplier.class);  
bind(CSVSupplier.class).to(BritishCSVSupplier.class);
```

In this manner, every dependency of `FlightSupplier` would be taken care by `CSVSupplier` and in turn `CSVSupplier` would be wired against the instance of `BritishCSVSupplier`.

## Instance bindings

A particular type could be bound to a specific instance of that type. This is a preferred strategy to inject the value objects. These objects do not have any dependencies and could be directly injected.

```
class ClientModule extends AbstractModule{
    @Override
    protected void configure() {
        bind(SearchRequest.class).toInstance(new SearchRequest());
    }
}
```

Any kind of objects which requires a lot of time to be instantiated should not be configured using `.toInstance()`. This could have an adverse effect on the startup time.

## Untargeted bindings

Untargeted bindings form an interesting part. These are actually indications to injector about a type so that the dependencies are prepared eagerly. Consider a case where in we declare something like this in `CSVSupplierModule`:

```
bind(String.class).toInstance("./flightCSV/");
```

The injector would eagerly prepare an instance of `String` class with a value of `./flightCSV/`. In case it receives a dependency injection scenario where in it requires, to inject an instance of `String`, it would inject this particular instance.



There arises a question: what would be the usefulness of such eagerly prepared dependencies? The answer lies in the fact that bindings could be created without specifying a target. This is useful for concrete classes and types annotated by `@ImplementedBy` or `@ProvidedBy`. In the next section of *Constructor bindings*, we are going to see a particular example where in an untargeted binding plays an essential role.

## Constructor bindings

This type of binding helps to bind a type to a constructor. This particular case arises when we cannot use constructor injection. This is a typical case when:

- We are using third-party classes
- Multiple constructors for a particular type are participating in dependency injection

In such a scenario, we have `toConstructor()` binding to our rescue. Here, we reflectively select our target constructor and handle the exception if the constructor is not found.

Let us take an example, consider replacing the attribute for `File` type which refers to the `csvFolder` in the method `loadCSV()` of `CSVSupplier` with an injected one. Simply we would create a reference to file in `CSVSupplier`:

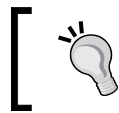
```
private File csvFolder;
```

Annotate setter with `@Inject` for setter injection:

```
@Inject
public void setCsvFolder(File csvFolder) {
    this.csvFolder = csvFolder;
}
```

Now, we need to provide a constructor binding for `File` type in `CSVSupplierModule`:

```
try {
    bind(File.class).
        toConstructor(
            File.class.getConstructor(String.class));
} catch (SecurityException e) {
    e.printStackTrace();
} catch (NoSuchMethodException e) {
    e.printStackTrace();
}
```



We choose the constructor reflectively; hence we need to handle checked exceptions thrown by the `Class.getConstructor` API.

Herein, we simply declare that for a type of `File`, we need to bind to a constructor of `java.io.File`, which accepts a `String` argument. Injector automatically resolves the invocation of the constructor and prepares an instance of `java.io.File` and injects accordingly.

How is the instance of `String` prepared to call the `File` specified constructor? In this case, the injector first looks up whether there are any eager dependencies without any targets available. If not, then it prepares an arbitrary instance of `String` and injects it.

In our case we have specified an untargeted binding while discussing the last topic. We've prepared an instance of `String` with a value of `./flightCSV/`. Now, injector actually uses this instance of `String` to be injected to `File` constructor.

## Binding annotations

There are certain occasions wherein we require multiple bindings of a similar type. For example, we would declare one more attribute of type `FlightSupplier` in `FlightEngine`. We would like to bind this reference to `XMLSupplier`. Now, we are facing a situation where in we need to inject two different implementations of a single interface.

```
private FlightSupplier csvFlightSupplier;

private FlightSupplier xmlFlightSupplier;
```

There are two ways to handle such situations:

- Using `@Named` annotations
- Using custom annotations

The annotation and the type together uniquely identify a binding. This pair of declaration is termed as key. The dependency resolution happens through this key.

Let us consider the first case. We just want to inject different values for different attributes of the same type. Just annotate the different field setters with `@Named` annotation specifying different String parameters which act as identifiers. For example, `xmlFlightSupplier` would be injected an `XMLFlightSupplier` dependency, the following way:

```
@Inject
public void setXmlFlightSupplier
    (@Named("xmlSupplier") FlightSupplier xmlFlightSupplier) {
    this.xmlFlightSupplier = xmlFlightSupplier;
}
```

Now for the binding part, in `FlightEngine` module, we need to write a binding.

```
bind(FlightSupplier.class)
    .annotatedWith(Names.named("xmlSupplier"))
    .toInstance(new XMLSupplier());
```

The injector searches for the `FlightSupplier` type annotated, with a named annotated argument as `xmlSupplier`. This particular dependency is resolved and injected to an instance of `XmlSupplier`. The important point is that though `@Named` annotation could be placed over method and fields, we need to explicitly annotate the argument of the method for injection to work.

Let us discuss the other possibility. We would like to develop our own annotation. Consider developing @CSV annotation for CSVSupplier. Here is the relevant code:

```
@BindingAnnotation @Target({ FIELD, PARAMETER, METHOD }) @
Retention(RUNTIME)
public @interface CSV {}
```

The relevant piece of code for injection inside FlightEngine would be to just annotate the injected parameter with @CSV:

```
@Inject
public FlightEngine(@CSV FlightSupplier flightSupplier) {
    this.csvFlightSupplier = flightSupplier;
}
```

The binding configuration in FlightEngineModule would be:

```
bind(FlightSupplier.class).
    annotatedWith(CSV.class).
    to(CSVSupplier.class);
```



The custom annotation developed involves the following things.

@Target and @Retention are standard annotations used while developing annotations in Java, which let us know to specify the target locations like field, method, parameter, and retention policy as RUNTIME respectively.

@BindingAnnotation is an annotation, used by Guice, which could be applied only to ANNOTATION\_TYPE. It basically makes sure that only one such annotation is applied at any injection point. Otherwise, Guice raises an error for the same.

## Binding constants

Following is an example where we inject a constant in FlightEngine:

```
bindConstant().
    annotatedWith(Names.named("maxResults")).
    to(10);
@Inject
public void setMaxResults(@Named("maxResults")int maxResults) {
    this.maxResults = maxResults;
}
```

`bindConstant()` provides a handy way to inject constants. It provides a reference to the `AnnotatedConstantBindingBuilder` interface, which in turn provides a reference to `ConstantBindingBuilder` interface via overloaded `annotatedWith()` API.

`ConstantBindingBuilder` interface provides overloaded `.to()` API, which is our key to binding constants. All primitive types, `String`, `Class`, and `Enum` are the acceptable parameters. Here, we could also provide corresponding wrapper types for primitives as parameters, since they would be autoboxed to call the corresponding API.

## Binding properties

While binding the constants, it's clumsy to declare them programmatically. A simpler way is to read them via the property files. Property files are one of the most convenient configuration option for reading database properties, file handling, and similar things. Consider injecting `csvPath` in `CSVSupplier` via the properties. Here in we need to provide `@Named()` annotation with a property. Following binding could be used to prepare bindings for all of the properties in a properties file. Have a look at the `CSVSupplierModule`:

```
Names.bindProperties(binder(), csvProperties());
```

Herein, we invoke the `bindProperties` method, `csvProperties` is our custom utility method, which is a reference to `csvSupplier`:

```
public Properties csvProperties(){
    Properties properties = new Properties();
    FileReader fileReader;
    try {
        fileReader =
            new FileReader(new File("./properties/csv.properties"));
        try {
            properties.load(fileReader);
        }catch (IOException e) {
            e.printStackTrace();
        }finally{
            try {
                fileReader.close();
            }catch (IOException e) {
                e.printStackTrace();
            }
        }
    }catch (FileNotFoundException e1) {
        e1.printStackTrace();
    }
    return properties;
}
```



## Just in time bindings

In case, there are no explicit bindings declared for an injection, Guice tries to create a just in time binding, in case there is no explicit binding available.

## By default constructors

By default, no argument constructors are invoked to prepare instances for injection. For instance, in our examples there is no explicit binding as how to create an instance of `Client`. However, a default constructor is invoked by the injector to return the instance of `Client`.

## @Inject annotated constructors

Constructors annotated with `@Inject` are also eligible for implicit bindings. The non-default constructor for `FlightEngine` is annotated with the `@Inject` annotation and is invoked to wire to the dependency in `Client`.

```
@Inject
public FlightEngine(@CSV FlightSupplier flightSupplier) {
    this.csvFlightSupplier = flightSupplier;
}
```

## Interfaces annotated with @ImplementedBy

An interface could be annotated with `@ImplementedBy` to declare implicitly, which implementation is to be injected for this interface.

Let us revisit the case of the `FlightSupplier` interface and its implementation as `CSVSupplier`. For injecting all references of the `FlightSupplier` interface the instance of `CSVSupplier` class, `@ImplementedBy` would suffice:

```
@ImplementedBy(CSVSupplier.class)
public interface FlightSupplier {
    Set<SearchResponse> getResults();
}
```

## Static injection

There are certain cases where in static attributes need to be dependency injected. Let us consider one such case. `FlightUtils` is a utility class providing a static method `parseDate` to parse a `String` in a given format and return the date. The format `String` is hardcoded in this API. Let us try to inject it statically.

Let us create a private static `String` attribute `dateFormat` and create its getter and setter. Mark the setter for injection with a `@Named` parameter.

```
@Inject
public static void setDateFormat(
    @Named("df")String dateFormat){
    FlightUtils.dateFormat = dateFormat;
}
```

Next, prepare a module for `FlightUtils`:

```
class FlightUtilityModule extends AbstractModule{
    @Override
    protected void configure() {
        bindConstant().annotatedWith(
            Names.named(dateFormat)).to(dd-MM-yy);
        requestStaticInjection(FlightUtils.class);
    }
}
```

Static injection comes into picture, (if requested) immediately once the injector instance is returned. Hence, we need not invoke the `.getInstance()` API on injector explicitly, for performing the static injection to happen.

## Summary

We have covered several features present in Guice, which help to cover majority cases of dependency injection. Mostly these revolve around various ways to bind objects to one another using explicit bindings. When it comes to defining configurations, explicit bindings should be preferred. For simple cases, just in time bindings suffice. In the next chapter we will delve into concepts like providers, scope, and binding collections.



# 3

## Diving Deeper in Guice

With basics of Guice covered, we are now ready to get into more challenging cases in the application development. These issues are addressed with the Guice features such as **Providers**, **Binding collections**, and **Scoping**. These are discussed in detail with several examples. Examples are available for these discussions in `flights_3_1` and `flights_3_2`.

### Going the extra mile with Providers

So far, we have worked with directly injecting dependencies. Yet, there are a few cases where injecting direct dependencies may not be an appropriate choice. Let's discuss such scenarios, and figure out how a `Factory` style `Provider` class provides a better way to inject dependencies.

### Need for a Provider

Guice provides an instance by invoking a suitable constructor. Whenever dependency injection happens, Guice requires all the instances be ready upfront to complete the wiring process. This is, however, undesirable in various situations. Consider `FlightSupplier`. Every time you require an instance of `Client`, all the dependencies would be wired to `CSVSupplier`. But effectively speaking, until a `processRequest()` invocation is made on `FlightEngine`'s instance, `CSVSupplier` doesn't come into picture.

Following are certain drawbacks of early instantiation:

- Late initialization may be required. Based on the client request, it would be unnecessary to parse all the `.csv` files. For example, if a request is now made with a filter for BA flights only; this means that only `BritishAirways.csv` needs to be parsed.

- An instance creation could throw an exception, and could delay or completely halt the process. For example, an instance of file is injected in csvFolder reference in CSVSupplier. This could throw an exception while instantiation.

Also, consider another case for the Value objects. Value objects, such as `SearchRequest` and `SearchResponse`, could also be dependency injected. Though if they were also wired early, as soon as the object graph preparation starts, their instances would be created beforehand.



A value object is a small object that represents a simple entity whose equality isn't based on identity. Two value objects are equal if they have same value, not necessarily being the same object. To know more about it, visit [http://en.wikipedia.org/wiki/Value\\_object](http://en.wikipedia.org/wiki/Value_object)

## Working of Provider

Usually, while binding, which is more or less a linked binding, we bind an interface to its, implementation. However, using Provider, we actually bind an interface to the implementation of the Provider interface.

Provider interface has only a single method. This method works as a factory method that Guice invokes in case it requires an instance of the class.

## Rolling your own Provider

Let's start with writing a Provider for CSVSupplier. Let's write an implementation for the same in package `org.packt.supplier.provider`.

```
class CSVSupplierProvider implements Provider<CSVSupplier>{
    @Override
    public CSVSupplier get() {
        CSVSupplier csvSupplier = new CSVSupplier();
        csvSupplier.setCsvFolder(new File("./flightCSV"));
        return csvSupplier;
    }
}
```

Next, we would be injecting a Provider instead of injecting the dependency itself. For this, the following changes should be done to `FlightEngineModule`.

```
bind(FlightSupplier.class).
    annotatedWith(CSV.class).
    toProvider(CSVSupplierProvider.class);
```

This configuration binds to a Provider rather than an instance. Now, Provider needs to be injected as per the configuration. The following changes are required in FlightEngine:

```
@Inject @CSV
private Provider<FlightSupplier> csvSupplierProvider;
```

Now, whenever we need an instance of CSVSupplier, we need to invoke the `csvSupplierProvider.get()` method. Please note the changes in `FlightEngine.processRequest()` API. Here, we are now referring to `csvSupplierProvider.get().getResults()` to fetch the results. For the sake of illustration, `CSVSupplier.loadCSVFiles()` is also refactored. Please refer to the same.

## Injecting dependencies in Providers

What if a provider itself has dependencies? Dependencies could be injected in Providers as well. Let's make a change to `CSVSupplierProvider`, add an attribute of type `String`, which is path to a directory with CSVFiles. We would inject the value in it. Following is the code:

```
private String csvFolderPath;

@Inject
public CSVSupplierProvider(
    @Named("csvFolderPath") String csvFolderPath) {
    this.csvFolderPath = csvFolderPath;
}
```

This could be bound to a `String` instance using following configuration in the `configure()` method of `FlightEngineModule`.

```
bind(String.class)
    .annotatedWith(Names.named("csvFolder"))
    .toInstance("./flightCSV");
```

The modified `get()` method of `CSVSupplierProvider` can now use the injected `csvFolderPath` attribute instead of a hardcoded value.

```
@Override
public CSVSupplier get() {
    CSVSupplier csvSupplier = new CSVSupplier();
    csvSupplier.setCsvFolder(new File(csvFolderPath));
    return csvSupplier;
}
```

## Advantages of Provider

Following are advantages of Provider:

- Multiple Providers could be configured to handle a case, where multiple instances are required.
- Delayed object creation. Instance creation is delayed until a request for the same is needed.
- A Provider could also be used like `@ImplementedBy` to suggest implicit bindings. Here, we need to annotate the interface with `@ProvidedBy`, which takes the argument of a class, which implements the Provider interface. Injector infers this as an implicit binding and invokes the `get()` method of provider to inject the dependency. For example:

```
@ProvidedBy(CSVSupplierProvider.class)
public interface FlightSupplier {
    Set<SearchResponse> getResults();
}
```

## @Provides

Annotation `@Provides` is an easy alternative to writing an implementation of Provider. In any of the modules, just provide a public method, which would return an instance of the class you are trying to provide. Consider a fictitious `JSONSupplier`, which implements `FlightSupplier`, and needs to be injected in `FlightEngine`.

Let's look at the injection part in `FlightEngine`.

```
private FlightSupplier jsonSupplier;

@Inject
public void setJsonSupplier(
    @Named("jsonSupplier") FlightSupplier jsonSupplier) {
    this.jsonSupplier = jsonSupplier;
}
```



Here we have used `@Named("jsonSupplier")` to distinguish it from other implementation of the `FlightSupplier`.

Let's look at the relevant code in `FlightEngineModule`, which acts a Provider implementation for `JSONSupplier`.

```
public class FlightEngineModule extends AbstractModule {
    @Provides
    @Named("jsonSupplier")
    public FlightSupplier provideJSONSupplier() {
        return new JSONSupplier();
    }
}
```

Guice picks up the `@Provides` annotation to create a provider method binding. It simply binds the return type to its returned value. The annotations, scopings, and so on are applied to the binding type as well.

## CheckedProviders

While suitable for most requirements, Providers have a problem. They don't have any provision for handling checked exceptions, and rely on the fact that the object initialization is free of exceptions. Also, any run-time exception thrown during the provisioning is wrapped up as a `ProvisionException`. For cases where in the checked exceptions are imminent during provisioning, `CheckedProvider` provides a way out.

`CheckedProvider` is an interface, which is a part of extension guice-throwingproviders.

Let's have a look at the interface:

```
public interface CheckedProvider<T> {
    T get() throws Exception;
}
```

It simply provides a convenient way to throw checked exceptions, which occur while provisioning. Let's create a hypothetical `ExcelSupplier` class to illustrate the case of throwing exceptions while provisioning:

```
public class ExcelSupplier implements FlightSupplier {
    public ExcelSupplier() throws NoExcelAvailableException {
        throw new NoExcelAvailableException();
    }
}
```



Here, we are deliberately throwing an exception during instantiation of the `ExcelSupplier`, which is a checked exception. Because it is not possible to implement a `Provider`, we would implement a `CheckedProvider`. Before that, we would extend the `CheckedProvider` in a custom interface in order to declare our own exceptions. Let's create an interface `ExcelCheckedProvider` interface, which extends `CheckedProvider`.

```
public interface ExcelCheckedProvider<T> extends CheckedProvider<T>{
    T get() throws NoExcelAvailableException;
}
```

Here, we extend the `CheckedProvider` to declare throwing of a particular subclass of exception. We could even throw multiple exceptions, if required.

Next, create an implementation of `ExcelCheckedProvider`:

```
public class ExcelCheckedSupplierProvider
implements ExcelCheckedProvider<FlightSupplier> {
    @Override
    public FlightSupplier get() throws NoExcelAvailableException {
        return new ExcelSupplier();
    }
}
```

We need to bind this implementation of `CheckedProvider` in `FlightEngineModule`:

```
ThrowingProviderBinder.create(binder()).
    bind(ExcelCheckedProvider.class, FlightSupplier.class).
        to(ExcelCheckedSupplierProvider.class);
```

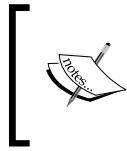
`ThrowingProviderBinder` builds a binding for a checked provider, and binds it to an implementation of the `CheckedProvider`.

Inject the `CheckedProvider` in `FlightEngine`:

```
@Inject
private ExcelCheckedProvider<FlightSupplier> excelCheckedProvider;
```

Now during usage, we need to write exception-handling code for the compile-time exception being thrown during provisioning.

```
try
{
    excelCheckedProvider.get().getResults();
} catch (NoExcelAvailableException e) {
    e.printStackTrace();
}
```



guice-throwingproviders also supply a ThrowingProvider interface like the CheckedProvider interface. This interface is marked as deprecated since the 3.0-rc2 release of guice-throwingproviders.

## @CheckedProvides

Just like @Provides, @CheckedProvides works in a similar fashion, just that method, which is annotated with @CheckedProvides could throw an exception. In this case, a method inside any module could be simply:

```
@CheckedProvides
public FlightSupplier get() throws NoExcelAvailableException {
    return new ExcelSupplier();
}
```

While using @Provides, the injection happens irrespective of using a provider or direct dependency injection. This, however, doesn't work with @CheckedProviders, because an invocation on the get() method could result in an exception. This is the key difference between @Provides and @CheckedProvides, wherein the earlier value could be directly injected without the Provider. Here, we need a place to handle the exception appropriately. Annotations(@Named or custom) or scoping, work apparently same in both the cases.


## AssistedInject

For Guice solutions like Providers, CheckedProviders seem to be a nice replacement of the Factory classes. Yet, there are certain cases where in Factories seem to be irreplaceable. Consider a case where in constructor for a class has arguments, and there is no default constructor available. The problem over here is very common with the value or domain objects. For example, if SearchRequest were modified a little, and default constructor is changed to the following:

```
public SearchRequest(String departureLocation,
String arrivalLocation, Date flightDate){
    this.departureLocation = departureLocation;
    this.arrivalLocation = arrivalLocation;
    this.flightDate = flightDate;
}
```

Now, in such a case, injecting the Provider and obtaining the value object instance with `provider.get()` would not be feasible. One solution could be implementing a factory and via it exposing the APIs to constructor. Then, we would inject the factory to prepare value objects.

Guice provides a simple solution for such a case. Using the extension `guice-assistedinject-3.0.jar`, the dynamic implementations of a factory interface could be directly injected and could be used for generating instances.

 Guice-assistedinject-3.0.jar is provided as a maven dependency in the project `pom.xml`.

Let's have a look at the factory interface first. We would need an implementation of the factory interface to be used to create instances of `SearchRequest`.

```
public interface SearchRQFactory {  
    SearchRequest create(String depLoc,String arrivLoc,Date  
        flightDate);  
}
```

A bound implementation, when injected, could be used as the following in `Client` and `FlightSearchAction`:

```
@Inject  
SearchRQFactory searchRQFactory;  
SearchRequest searchRQ =  
    searchRQFactory.create("AMS", "MAD", flightDate);
```

For accomplishing this, all we need to do is annotate the following APIs and parameters with the `@AssistedInject` and `@Assisted` annotations.

Let's see these in action. First of all, we need to annotate the `SearchRequest` constructor with the `@AssistedInject` annotation.

```
@AssistedInject  
public SearchRequest(  
    @Assisted("depLoc") String departureLocation,  
    @Assisted("arrivLoc") String arrivalLocation,  
    @Assisted Date flightDate){  
    this.departureLocation = departureLocation;  
    this.arrivalLocation = arrivalLocation;  
    this.flightDate = flightDate;  
}
```

This specifies that the `SearchRequest` constructor would be assisted in injection via a factory. The factory interface needs to be created as well, with following annotations:

```
package org.packt.client;

import java.util.Date;

import com.google.inject.assistedinject.Assisted;

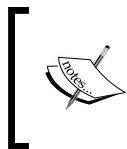
public interface SearchRQFactory {
    SearchRequest create(
        @Assisted("depLoc") String depLoc,
        @Assisted("arrivLoc") String arrivLoc,
        Date flightDate);
}
```

Now, it's time to bind the `SearchRQFactory` with a `FactoryModuleBinder`. A `FactoryModuleBinder` provides a factory that combines the invoker's argument to create objects. Along with invoking arguments, there could also be injector-supplied values. In our case, we don't have any injector supplied values, but they could be present as arguments. They need to be annotated with the `@Inject` annotation to facilitate injection. Let's have a quick look at the binding in `ClientModule`.

```
import com.google.inject.assistedinject.FactoryModuleBuilder;

FactoryModuleBuilder fb = new FactoryModuleBuilder();
install(fb.build(SearchRQFactory.class));
```

This simply builds a factory for interface `SearchRQFactory` and installs it as a module.



`@Assisted` provides a way to inject parameters from a value coming from arguments to a factory method. `@Assisted` accepts a value identifier "name" to identify among parameters of the same type. In case of distinct type of parameters, the "name" value is not required.

## Binding collections

We saved binding collections for this chapter, because this is something, which requires use of Providers, we will introduce `MultiBinder` for binding Guice managed objects.

## Using TypeLiteral

Let's start with a simple example, where in we need to inject objects that are not managed by Guice. Let's try to inject a `Set<String>` messages in `FlightEngine`. Define an attribute, and annotate the setter with injector.

```
@Inject
public void setMessages(Set<String> messages) {
    this.messages = messages;
}
```

Next, we need to write a provider for the same:

```
public class MessageProvider implements Provider<Set<String>> {

    @Override
    public Set<String> get() {
        Set<String> messageSet = new HashSet<String>();
        messageSet.add("Hi Client");
        messageSet.add("Bye Client");
        return messageSet;
    }
}
```

The provider should be bound using a `TypeLiteral`:

```
bind(new TypeLiteral<Set<String>>(){}).
    toProvider(MessageProvider.class).
    in(Singleton.class);
```

Let's examine `TypeLiteral`. It helps us to represent generic types. All we do is to create a subclass of it, to enable retrieval of type information at runtime. Here, an empty anonymous class which subclasses `TypeLiteral<Set<String>>` suffices in our case.

## Using MultiBinder

When it comes to inject collections of Guice managed objects, `MultiBinder` shines. It provides a suitable way to bind multiple values. These values are later injected as a complete collection.

The best case to describe a use of `MultiBinder` would be to tweak our example a little. We have different implementations of `FlightSupplier` interface as `CSVSupplier` and `XMLSupplier`. Let's prepare a new implementation of `FlightSupplier` as `JSONSupplier`. Now, what we intend is to refer to both `JSONSupplier` and `XMLSupplier`. In this case, we prefer to have them available as a collection of `FlightSupplier`.

Let's prepare a collection of `FlightSupplier` and `Set<FlightSupplier>` in `FlightEngine`.

```
private Set<FlightSupplier> extraSuppliers;
@Inject
public void setExtraSuppliers(Set<FlightSupplier> extraSuppliers) {
    this.extraSuppliers = extraSuppliers;
}
```

Now, let's use `MultiBinder` in our `FlightEngineModule` to configure collection binding.

```
MultiBinder<FlightSupplier> multiBinder =
    MultiBinder.newSetBinder(binder(), FlightSupplier.class);
multiBinder.addBinding().to(XMLSupplier.class).asEagerSingleton();
multiBinder.addBinding().toInstance(new JSONSupplier());
```



MultiBinder, is available as an extension API of Guice, `guice-multibindings-3.0.jar`. This is declared as a dependency in `pom.xml`.



## Scoping

Scoping is a way to control the lifetime of an object. Guice upon request creates the object and injects the dependencies. A classic example could be of a `Singleton` class, where in every reference must lead to the same object. Another such example could be a `Value` object, such as `SearchRequest` and `SearchResponse`, wherein every `processRequest` invocation should be done with a new `SearchRequest` instance.

## Singletons with Guice

`Singleton` is a design pattern used to create a single object, which would be referred by all references throughout the application life cycle. It is basically a provision to always return the reference to a single object.

Generally singletons are implemented via static factory methods, which have synchronized blocks to ensure that reference to single object is returned. Singletons are also designed using inner private classes, which are available only on request, and are not loaded early.

This approach works perfectly well when `Singleton` itself is independent of any dependencies. Otherwise designing such a singleton with static factories taking care of parameter(s) while object creation becomes imperatively clumsy.

Guice beautifully solves this problem using Scope. There are two approaches to define a class as singleton. Let's make CSVSupplier a Singleton.

We need these next to our bindings to define it as a Singleton. This code goes in FlightEngineModule.

```
bind(FlightSupplier.class) .
  annotatedWith(CSV.class) .to(CSVSupplier.class) .
  in(Scopes.SINGLETON);
```

The same results could be achieved using:

```
bind(FlightSupplier.class) .
  annotatedWith(CSV.class) .to(CSVSupplier.class) .
  in(Singleton.class);
```

The Scope could also be applied by annotating the classes directly with the @Singleton annotation.

```
@Singleton
public class CSVSupplier implements FlightSupplier{..}
```

Any of the three implementations works fine. However, Guice provides an additional way to bind to a scope annotation.

## Eager Singletons

Singletons are created eagerly in a production environment. This is, however, not true in a development environment. In case of a development environment the initialization is lazy. The lazy initialization is more suitable for frequent changes and testing. However, in case we need a snappy experience even in development environment `.asEagerSingleton()` should be used.

```
bind(FlightSupplier.class) .
  annotatedWith(Names.named("xmlSupplier")) .
  to(XMLSupplier.class) .asEagerSingleton();
```

Initialization problems are revealed quickly by using the Eager Singleton approach. This is the foremost reason why default approach in production is to initialize singletons eagerly.



In case we need to have a lazy initialization approach, even in production, we can use the `@LazySingleton` annotation.

How to switch environments from development to production and vice versa? Stage could be set using enum `com.google.inject.Stage`. There are two APIs available in `Guice.class` to specify stage while preparing the injector.

```
public static Injector createInjector(Stage stage, Module... modules)

public static Injector createInjector(Stage stage, Iterable<? extends
Module> modules)
```

If not specified, the default stage is set to `Stage.DEVELOPMENT`.

## Custom Scopes

Most of the instances for classes we have created so far fall in two categories. The one provided as a Singleton or the other provided as an instance by a provider. However, owing to different circumstances, we need custom scoping, where a developer could strategize, when to create a new instance for a class. Guice provides a way to declare custom scopes and their usage via providers.

### Motivation to write a custom scope

Let's consider a situation where the `.csv` files in the `csvFolder` are updated often. In this case, we need to develop a mechanism so that after a certain time, our provider for `CSVSupplier` creates a new instance. To implement this use case, we would be doing a poll wherein after every 60 seconds, a new `CSVSupplier` is instantiated by the provider. Now, implementing such a case in our business logic could result in mixing of concerns. Search logic shouldn't care whether time to poll has elapsed or not. To handle such a case, when a previously instantiated object went out of scope or not, we need to actually define what a scope would be in our use case.

To put it simply, a scope of a `csvSupplier` instance would be as long as the interval has elapsed or not. Once interval is elapsed, the previously created `csvSupplier` instance (which was supplied by `csvSupplierProvider`) is now out of scope.



## Modifying our Provider

Before we implement a Scope, we need to do a few changes to our provider. Let's revisit our CSVSupplierProvider. Whenever `get()` API is invoked on CSVSupplierProvider a new instance of CSVSupplier is prepared and returned. We need to change a few conditions now. Let's look at the new CSVSupplierProvider. We'll start with the `get()` API.

```
@Override
public CSVSupplier get() {
    if(csvSupplier == null || !inScope())
        newSupplier();
    return csvSupplier;
}
```

Here, we check whether an instance of `csvSupplier` is available or not, if not we invoke `newSupplier()` for the same.

```
public void newSupplier(){
    csvFolder = new File("./flightCSV");
    csvSupplier = new CSVSupplier();
    timeStamp = csvFolder.lastModified();
    csvSupplier.setCsvFolder(csvFolder);
}
```

Here, we do following things:

- Prepare an instance of `File`, which refers to `csvFolder` directory.
- Prepare an instance of `CSVSupplier`.
- Store the last modified timestamp of `csvFolder` directory.

We also keep the references locally, for the `File` object and `CSVSupplier`. Now, we have two APIs, which would be used by our custom scope.

```
public boolean inScope(){
    Long currentTimestamp = System.currentTimeMillis();
    // Let it go out of scope after 60 seconds.
    return ((currentTimestamp - timeStamp) > 60000 ? false : true);
}
```

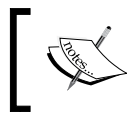
`inScope()` is a public API to calculate whether we are in scope or not.

This would cause a new instance to be prepared by the `get()` API of the CSVSupplierProvider.

## Defining our own scope

Let's implement the Scope interface and write our scope detection logic in it. A scope should return the reference to the provider. The overridden method has two parameters, a reference to binding key and a reference to an **unscoped** provider. A binding key consists of an injection type and an optional annotation. Reference unscoped locates an instance when one doesn't already exist in this scope.

A Scope comes into picture during the binding phase. When the `scope()` API is invoked, it brings provider into picture. We instantiate our own provider here and return it.



We will discuss custom scopes in more detail in *Chapter 4, Web Development with Google Guice*. Web application development involves various scenarios, which are obvious candidates for custom scoping.

```
package org.packt.scope;
import com.google.inject.*;

public class CSVScope implements Scope {
    @Override
    public <T> Provider<T> scope(Key<T> key, Provider<T> unscoped) {
        return (Provider<T>) new CSVSupplierProvider();
    }
}
```

## Changes in Module

The changes in modules are as follows:

### Direct binding

The following changes are required in `FlightEngineModule`, for direct binding of a scope to an instance or a provider.

```
bind(FlightSupplier.class).
    annotatedWith(CSV.class).
        toProvider(CSVSupplierProvider.class).in(new CSVScope());
```

## Binding to an annotation

Here, we will create a scope annotation, and bind to annotation rather than binding directly to Provider or an instance. This helps us in two ways:

- Switching between scopes or developing multiple scope provider combination would be easy
- No binding between scope and implementation or provider

## Developing an annotation for a scope

@ScopeAnnotation annotation is used to define a scope-based binding. Following is the @InScope annotation, which is annotated by it.

```
package org.packt.scope;

@Target({ FIELD, TYPE}) @Retention(RUNTIME)
@ScopeAnnotation
public @interface InScope {}
```

Following is the InScope annotation, which is annotated by @ScopeAnnotation. This annotation is used to define a scope-based binding. Next, we just need to change our module.

```
bind(FlightSupplier.class).
    annotatedWith(CSV.class).
    toProvider(CSVSupplierProvider.class).in(InScope.class);
bindScope(InScope.class, new CSVScope());

@Inject @CSV
private Provider<FlightSupplier> csvSupplierProvider;
```

Now, there is no binding between a Scope and a Provider or implementation.



Scope-related configuration could be moved to separate modules. While binding a scope to a Scope annotation, we wrote the configurations in `FlightEngineModule`. This could actually be separated, because binding a scope is not related to its provider or implementation any way. Let's roll out a separate module for it.

```
public class CSVScopeModule extends AbstractModule
{
    @Override
    protected void configure() {
        bindScope(InScope.class, new CSVScope());
    }
}
```

## Observations

Clearly, we could see that custom scope development is a very isolated case and finds its place rarely in the regular dependency injection scenarios. For most of the cases, Guice-provided annotations for handling scope should suffice. Custom scope development is therefore not suggested.

In the next chapter, we will see various Guice provided scopes to handle scoping in web application development.

## Summary

In this chapter, we have covered various features which are provided by Guice to handle pretty challenging cases in application design. Developing applications with Providers and scoping Guice managed objects was discussed thoroughly. We also saw how to bind collections using Guice. In the next chapter, we will develop web applications using Guice. We will look at how to write Servlets, JSPs in a Guice managed environment.



# 4

## Guice in Web Development

After making a deep dive in Guice, we are all ready to put it to some practical use. Here in, we would develop a web application in Java using servlets and JSPs, and will see how Guice makes it simple to wire dependencies and helps to achieve separation of concerns easily. It even provides a programmatic approach to configure routing to servlets and JSPs while avoiding declarative approach in `web.xml`. As part of our learning we will fit flight search functionality in a web application. It is assumed that the reader is familiar with JSP, servlet development, and knows how to deploy a WAR file to a web container. Use of Tomcat to deploy the examples is suggested. The example code for this chapter is present in `chapter_4/flightsweb`.

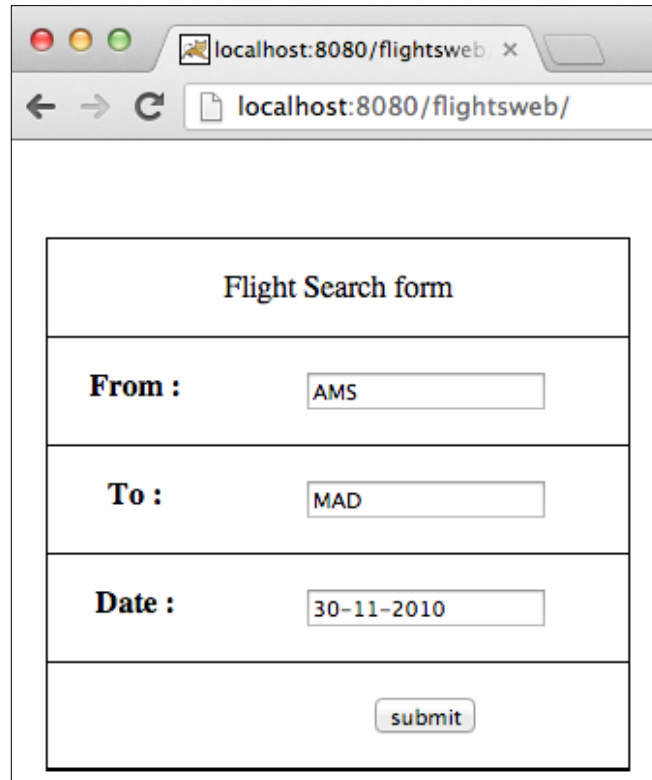


We also need an extension of Guice for web development, `guice-servlet-3.0`. As usual, the dependency for this has been declared in `pom.xml` and once packaged it would be available in **WAR (web application archive)** also.

### Structure of flightsweb application

Our application has two servlets: `IndexServlet`, which is a trivial example of forwarding any request, mapped with `"/` to `index.jsp` and `FlightServlet`, which processes the request using the functionality we developed in the previous chapters and forwards the response to `response.jsp`. Here in, we simply declare the `FlightEngine` and `SearchRequest` as the class attributes and annotate them with `@Inject`. `FlightSearchFilter` is a filter with the only responsibility of validating the request parameters. `Index.jsp` is the landing page of this application and presents the user with a form to search the flights, and `response.jsp` is the results page.

The flight search form will look as shown in the following screenshot:



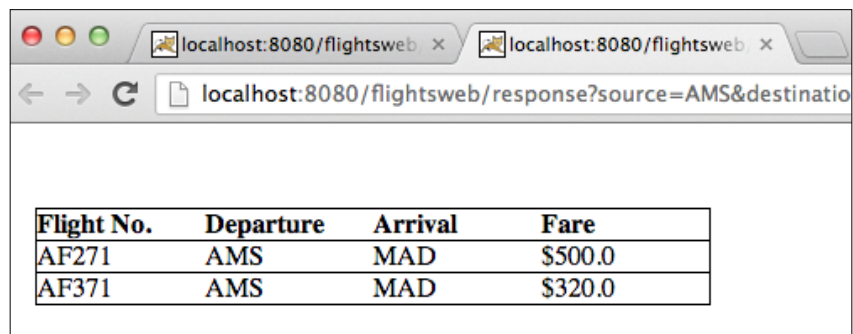
Flight Search form

**From :**

**To :**

**Date :**

The search page would subsequently lead to the following result page.



Flight No.	Departure	Arrival	Fare
AF271	AMS	MAD	\$500.0
AF371	AMS	MAD	\$320.0

In order to build the application, we need to execute the following command in the directory `chapter_4/flightsweb`, where the `pom.xml` file for the project resides:

```
shell> mvn clean package
```

The project for this chapter being a web application project compiles and assembles a WAR file, `flightsweb.war` in the target directory. We could deploy this file to TOMCAT.

## Using GuiceFilter

Let's start with a typical web application development scenario. We need to write a JSP to render a form for searching flights and subsequently a response JSP page. The search form would post the request parameters to a processing servlet, which processes the parameters and renders the response.

Let's have a look at `web.xml`. A `web.xml` file for an application intending to use Guice for dependency injection needs to apply the following filter:

```
<filter>
  <filter-name>guiceFilter</filter-name>
  <filter-class>com.google.inject.servlet.GuiceFilter</filter-class>
</filter>

<filter-mapping>
  <filter-name>guiceFilter</filter-name>
  <url-pattern>/*</url-pattern>
</filter-mapping>
```

It simply says that all the requests need to pass via the Guice filter. This is essential since we need to use various servlet scopes in our application as well as to dispatch various requests to injectable filters and servlets. Rest any other servlet, filter-related declaration could be done programmatically using Guice-provided APIs.

## Rolling out our ServletContextListener interface

Let's move on to another important piece, a servlet context listener for our application. Why do we need a servlet context listener in the first place? A servlet context listener comes into picture once the application is deployed. This event is the best time when we could bind and inject our dependencies.



Guice provides an abstract class, which implements `ServletContextListener` interface. This class basically takes care of initializing the injector once the application is deployed, and destroying it once it is undeployed. Here, we add to the functionality by providing our own configuration for the injector and leave the initialization and destruction part to super class provided by Guice. For accomplishing this, we need to implement the following API in our sub class:

```
protected abstract Injector getInjector();
```

Let's have a look at how the implementation would look like:

```
package org.packt.web.listener;

import com.google.inject.servlet.GuiceServletContextListener;
import com.google.inject.servlet.ServletModule;

public class FlightServletContextListener extends
    GuiceServletContextListener {
    @Override
    protected Injector getInjector() {
        return Guice.createInjector(
            new ServletModule(){
                @Override
                protected void configureServlets() {
                    // overridden method contains various
                    // configurations
                }
            });
    }
}
```

Here, we are returning the instance of injector using the API:

```
public static Injector createInjector(Module... modules)
```

Next, we need to provide a declaration of our custom `FlightServletContextListener` interface in `web.xml`:

```
<listener>
  <listener-class>
    org.packt.web.listener.FlightServletContextListener
  </listener-class>
</listener>
```

## ServletModule – the entry point for configurations

In the argument for modules, we provide a reference of an anonymous class, which extends the class `ServletModule`. A `ServletModule` class configures the servlets and filters programmatically, which is actually a replacement of declaring the servlet and filters and their corresponding mappings in `web.xml`.

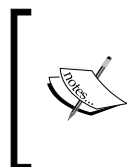
Why do we need to have a replacement of `web.xml` in the first place? Think of it on different terms. We need to provide a singleton scope to our servlet. We need to use various web scopes like `RequestScope`, `SessionScope`, and so on for our classes, such as `SearchRequest` and `SearchResponse`. These could not be done simply via declarations in `web.xml`. A programmatic configuration is far more logical choice for this. Let's have a look at a few configurations we write in our anonymous class extending the `ServletModule`:

```
new ServletModule() {
    @Override
    protected void configureServlets() {
        install(new MainModule());
        serve("/response").with(FlightServlet.class);
        serve("/").with(IndexServlet.class);
    }
}
```

A servlet module at first provides a way to install our modules using the `install()` API. Here, we install `MainModule`, which is reused from the previous chapter. Rest all other modules are installed from `MainModule`.

## Binding language

`ServletModule` presents APIs, which could be used for configuring filters and servlets. Using these expressive APIs known as EDSL, we could configure the mappings between servlets, filters, and respective URLs.



Guice uses an embedded domain specific language or EDSL to help us create bindings simply and readably. We are already using this notation while creating various sort of bindings using the `bind()` APIs. Readers could refer to the Binder javadoc, where EDSL is discussed with several examples.

## Mapping servlets

Here, following statement maps the `/response` path in the application to the `FlightServlet` class's instance:

```
serve("/response").with(FlightServlet.class);
```

`serve()` returns an instance of `ServletKeyBindingBuilder`. It provides various APIs, using which we could map a URL to an instance of servlet. This API also has a variable argument, which helps to avoid repetition. For example, in order to map `/response` as well as `/response-quick`, both the URLs to `FlightServlet.class` we could use the following statement:

```
serve("/response", "/response-quick").with(FlightServlet.class);
```

`serveRegex()` is similar to `serve()`, but accepts the regular expression for URL patterns, rather than concrete URLs. For instance, an easier way to map both of the preceding URL patterns would be using this API:

```
serveRegex("^response").with(FlightServlet.class);
```

`ServletKeyBindingBuilder.with()` is an overloaded API. Let's have a look at the various signatures.

```
void with(Class<? extends HttpServlet> servletKey);  
void with(Key<? extends HttpServlet> servletKey);
```

To use the key binding option, we will develop a custom annotation `@FlightServe`. `FlightServlet` will be then annotated with it. Following binding maps a URL pattern to a key:

```
serve("/response").with(Key.get(HttpServlet.class, FlightServe.class));
```

Since this, we need to just declare a binding between `@FlightServe` and `FlightServlet`, which will go in modules:

```
bind(HttpServlet.class).  
    annotatedWith(FlightServe.class).to(FlightServlet.class)
```



What is the advantage of binding indirectly using a key? First of all, it is the only way using which we could separate an interface from an implementation. Also it helps us to assign scope as a part of the configuration. A servlet or a filter must be at least in singleton. In this case we can assign scope directly in configuration. The option of annotating a filter or a servlet with `@Singleton` is also available, although.

Guice 3.0 provides following overloaded versions, which even facilitate providing initialization parameters and hence provide type safety.

```
void with(HttpServlet servlet);
void with(Class<? extends HttpServlet> servletKey, Map<String, String>
initParams);
void with(Key<? extends HttpServlet> servletKey, Map<String, String>
initParams);
void with(HttpServlet servlet, Map<String, String> initParams);
```

An important point to be noted here is that `ServletModule` not only provides a programmatic API to configure the servlets, but also a type-safe idiomatic API to configure the initialization parameters. It is not possible to ensure type safety while declaring the initialization parameters in `web.xml`.

## Mapping filters

Similar to the servlets, filters could be mapped to URL patterns or regular expressions. Here, the `filter()` API is used to map a URL pattern to a Filter. For example:

```
filter("/response").through(FlightSearchFilter.class);
```

`filter()` returns an instance of `FilterKeyBindingBuilder`.

`FilterKeyBindingBuilder` provides various APIs, using which we can map a URL to an instance of filter. `filter()` and `filterRegex()` APIs take exactly the same kind of arguments as `serve()` and `serveRegex()` does when it comes to handling the pure URLs or regular expressions.

Let's have a look at `FilterKeyBindingBuilder.through()` APIs. Similar to `ServletKeyBindingBuilder.with()` it also provides various overloaded versions:

```
void through(Class<? extends Filter> filterKey);
void through(Key<? extends Filter> filterKey);
```

Key mapped to a URL, which is then bound via annotation to an implementation could be exemplified as:

```
filter("/response").
through(Key.get(Filter.class, FlightFilter.class));
```

The binding is done through annotation. Also note, that the filter implementation is deemed as singleton in scope.

```
bind(Filter.class).
    annotatedWith(FlightFilter.class).
    to(FlightSearchFilter.class).in(Singleton.class);
```

Guice 3.0 provides following overloaded versions, which even facilitate providing initialization parameters and provide type safety:

```
void through(Filter filter);
void through(Class<? extends Filter> filterKey, Map<String, String>
    initParams);
void through(Key<? extends Filter> filterKey, Map<String, String>
    initParams);
void through(Filter filter, Map<String, String> initParams);
```

Again, these type safe APIs provide a better configuration option than declaration driven `web.xml`.

## Web scopes

Aside from dependency injection and configuration facilities via programmable APIs, Guice provides feature of scoping various classes, depending on their role in the business logic.

As we saw, while developing the custom scope in *Chapter 3, Diving Deeper in Guice*, a scope comes into picture during binding phase. Later, when the scope API is invoked, it brings the provider into picture. Actually it is the provider which is the key to the complete implementation of the scope. Same thing applies for the web scope.

## @RequestScoped

Whenever we annotate any class with either of servlet scopes like `@RequestScoped` or `@SessionScoped`, call to scope API of these respective APIs are made. This results in eager preparation of the `Provider<T>` instances. So to harness these providers, we need not configure any type of binding, as these are implicit bindings. We just need to inject these providers where we need the instances of respective types.

Let us discuss various examples related to these servlet scopes. Classes scoped to `@RequestScoped` are instantiated on every request. A typical example would be to instantiate `SearchRequest` on every request. We need to annotate the `SearchRQ` with the `@RequestScoped`.

```
@RequestScoped
public class SearchRequest { .....}
```

Next, in `FlightServlet` we need to inject the implicit provider:

```
@Inject
private Provider<SearchRequest> searchRQProvider;
```

The instance could be fetched simply by invoking the `.get()` API of the provider:

```
SearchRequest searchRequest = searchRQProvider.get();
```

## @SessionScoped

The same case goes with `@SessionScoped` annotation. In `FlightSearchFilter`, we need an instance of `RequestCounter` (a class for keeping track of number of requests in a session). This class `RequestCounter` needs to be annotated with `@SessionScoped`, and would be fetched in the same way as the `SearchRequest`. However the `Provider` takes care to instantiate it on every new session creation:

```
@SessionScoped
public class RequestCounter implements Serializable{.....}
```

Next, in `FlightSearchFilter`, we need to inject the implicit provider:

```
@Inject
private Provider<RequestCounter> sessionCountProvider;
```

The instance could be fetched simply by invoking the `.get()` API of the provider.

## @RequestParameters

Guice also provides a `@RequestParameters` annotation. It could be directly used to inject the request parameters. Let's have a look at an example in `FlightSearchFilter`. Here, we inject the provider for type `Map<String, String[]>` in a field:

```
@Inject
@RequestParameters
private Provider<Map<String, String[]>> reqParamMapProvider;
```

As the provider is bound internally via `InternalServletModule` (Guice installs this module internally), we can harness the implicit binding and inject the Provider.

An important point to be noted over here is that, in case we try to inject the classes annotated with `ServletScopes`, like `@RequestScoped` or `@SessionScoped`, outside of the `ServletContext` or via a non HTTP request like RPC, Guice throws the following exception:

```
SEVERE: Exception starting filter guiceFilter
com.google.inject.ProvisionException: Guice provision errors:
Error in custom provider, com.google.inject.OutOfScopeException:
Cannot access scoped object. Either we are not currently inside an
HTTP Servlet request, or you may have forgotten to apply com.google.
inject.servlet.GuiceFilter as a servlet filter for this request.
```

This happens because the Providers associated with these scopes necessarily work with a `ServletContext` and hence it could not complete the dependency injection. We need to make sure that our dependencies annotated with `ServletScopes` come into the picture only when we are in `WebScope`.



Another way in which the scoped dependencies could be made available is by using the `injector.getInstance()` API. This however requires that we need to inject the injector itself using the `@Inject` injector in the dependent class. This is however not advisable as it is mixing dependency injection logic with the application logic. We need to avoid this approach.

## Exercising caution while scoping

Our examples illustrate cases where we are injecting the dependencies with narrower scope in the dependencies of wider scope. For example, `RequestCounter` (which is `@SessionScoped`) is injected in `FlightSearchFilter` (which is a singleton).

This needs to be very carefully designed, as in when we are absolutely sure that a narrowly scoped dependency should be always present else it would create a problem. It basically results in scope widening, which means that apparently we are widening the scope of `SessionScoped` objects to that of singleton scoped object, the servlet. If not managed properly, it could result into memory leaks, as the garbage collector could not collect the references to the narrowly scoped objects, which are held in the widely scoped objects.

Sometimes this is unavoidable, in such a case we need to make sure we are following two basic rules:

- Injecting the narrow scoped dependency using Providers. By following this strategy, we never allow the widely scoped class to hold the reference to the narrowly scoped dependency, once it goes out of scope. Do not get the injector instance injected in the wide scoped class instance to fetch the narrow scoped dependency, directly. It could result in hard to debug bugs.
- Make sure that we use the dependent narrowly scoped objects in APIs only. This lets these to live as stack variables rather than heap variables. Once method execution finishes, the stack variables are garbage collected. Assigning the object fetched from the provider to a class level reference could affect garbage collection adversely, and result in memory leaks.

Here, we are using these narrowly scoped dependencies in APIs: `doGet()` and `doFilter()`. This makes sure that they are always available.

Contrarily, injecting widely scoped dependencies in narrowly scoped dependencies works well, for example, in a `@RequestScoped` annotated class if we inject a `@SessionScoped` annotated dependency, it is much better since it is always guaranteed that dependency would be available for injection and once narrowly scoped object goes out of scope it is garbage collected properly.



## Summary

We retrofitted our flight search application in a web environment. In doing so we learned about many aspects of the integration facilities Guice offers us:

- We learned how to set up the application to use dependency injection using `GuiceFilter` and a custom `ServletContextListener`.
- We saw how to avoid servlet, filter mapping in `web.xml`, and follow a safer programmatic approach using `ServletModule`.
- We saw the usage of various mapping APIs for the same and also certain newly introduced features in Guice 3.0.
- We discussed how to use the various web scopes.

This chapter has served as a primer to bring Guice extension for servlets for dependency injection in web development.

In next chapter we will explore Guice extension, which provides integration between Struts2 and Guice 3. While doing so we would redevelop the application using Struts2.

# 5

## Integrating Guice with Struts 2

This chapter demonstrates integration of Guice 3 with **Struts 2**. It is an action oriented, web framework based on MVC architecture. Guice integrates easily to provide dependency injection facility into the application developed. We will discuss the special modules and classes, which accomplish the task of integration. On top of it we would retrofit our application developed in the last chapter to be based on Struts 2. While doing so we will avoid writing servlets and filters and instead use Struts 2 provided classes.

It is desired that the reader is familiar with Struts 2, as primarily we are interested in injecting dependencies in Struts 2-based applications and not exploring Struts 2 itself. However, for the sake of completion of the subject matter, we will begin with a gentle introduction of the framework and go forward.



To begin development with Struts 2 we need `struts2-core-2.3.14.jar` file in our classpath. Also, for integrating it along with Guice, we need an extension of Guice for Struts2, `guice-struts-2.3.0.jar`. These two files along with various dependencies are listed in `pom.xml`, hence need not be separately downloaded. Readers could simply execute the maven command to package the WAR, and the necessary dependencies would be downloaded and packaged in the WAR file.

## Introducing Struts 2

Struts 2 is an action-oriented MVC framework for web application development. It is based on the philosophy that framework should be action-oriented and loosely coupled with application logic. For example, a request handling should be an action itself and need not involve interacting with pieces which are a part of the framework. Unlike other frameworks, which impose conditions like extending from a class or implementing an interface to handle requests, Struts 2 defies such behavior. Instead, it provides various convenient interfaces or classes that could be used as per our application requirements and upfront coupling is not required. Struts 2 also defies the use of special classes, which are used to convey the result objects from model to view. Against this, Struts 2 provides the **OGNL** and **ValueStack**, which help to make available the member of `Action` class variables in the view.



`ValueStack` is a `ThreadLocal` object created to cater a request. For every request a different `ValueStack` is created. It comprises of two parts.

First is an **ObjectStack**, where objects are created and stored for processing a request. These are named objects, such as `Application`, `Session`, and `Request`, which are stored prior to request specific objects. Later request specific objects, such as `Model` and `Action` are created and stored. Temporary objects, which are created during execution of a JSP page are also stored over here.

Second is an **ActionContextMap**, which holds various maps related to an application. An example is an `AttributeMap`. It is a map of Request attributes, Session attributes, and Application attributes. Using an attribute map indirectly via an `ActionContextMap`, we can search for any attribute present in a request, session or application attribute map.

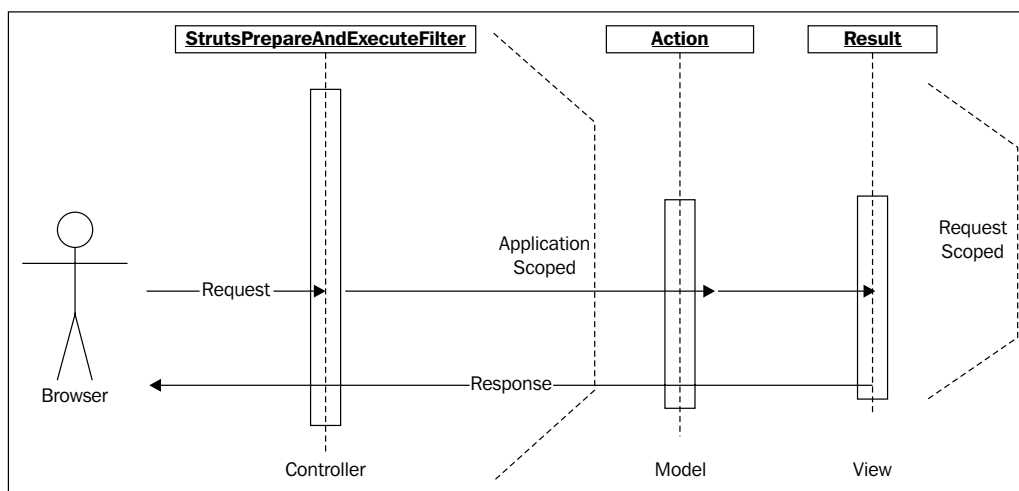
OGNL stands for Object Graph Navigation Language. It is similar to JSP Expression language and based on the idea of having a root object within the context. The properties of this root object can be referenced using the markup notation. An `ActionContextMap` is created by framework to act as a context. `ActionContextMap` consists of various objects, such as `Application`, `Session`, `ValueStack`, `Request`, `Parameters`, and `Attributes`.

OGNL assists in data transfer and type conversion. Its rich expression syntax allows easy referencing and manipulation of objects on `ValueStack`.

There are a few important characteristics of Struts 2, which facilitate application development greatly. These are:

- Interceptors for layering cross cutting concerns away from the Action logic
- Annotation-based configuration, which eliminates a lot of redundant configuration details
- A tag-based MVC API, which supports modifiable and reusable components

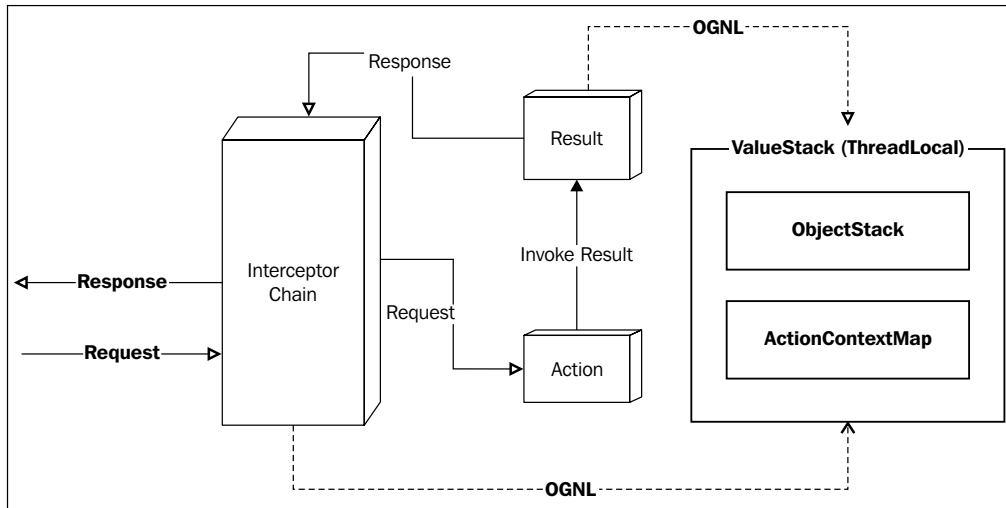
MVC implementation in Struts 2 is fairly simple. Have a look at the following diagram for better understanding:



`StrutsPrepareAndExecuteFilter` acts as a controller. The essential purpose it serves is to inspect each incoming request to determine which action should handle the request. Once identified it passes on the control to the particular action instance.

`Action`, which is a model of application, serves two important functions. It encapsulates all the business logic needed to accomplish the request in a single unit of work. Secondly, it indicates what view to serve as per the request conclusion. For example, it could route to different views based on the processing of request; an error page for handling exceptions while a response page on successful request completion.

View is the presentation component of the application development. It could be a JSP page, an XSLT handling an example response, a FreeMarker, or a Velocity template. The working of a Struts 2 application could be summarized using the following diagram:



A chain of interceptors from Struts 2 intercepts any request to create a ValueStack. Once a ValueStack is prepared it is used to store and manipulate request-related objects like Model and Action, via ActionContextMap. The result, which could be a JSP page, manipulates and refers to ValueStack using OGNL and finally renders the response, which also passes back through the interceptor chain. In this way, a complete request-response cycle is handled.

## Guice 3 and Struts 2 integration

There are several artifacts provided by guice-struts-2.3.0 extension to assist in implementing Guice 3 as the dependency injection provider. Let's examine them.

### Struts2GuicePluginModule

Struts2GuicePluginModule needs to be installed before any of our modules are installed. This module basically requests the static injection of the Struts2Factory class. Struts2Factory uses the injector later to prepare the instances of the requested object.

## Struts2Factory

The `Struts2Factory` class is a sub class of `com.opensymphony.xwork2.ObjectFactory`. `ObjectFactory` is responsible for building the core framework objects in Struts 2. `ObjectFactory.buildBean(class java.util.map)` method is used to build all the core framework objects like interceptors, actions, and results. `Struts2Factory` overrides this method and requests their instance creation using injector. This instance of `Struts2Factory` as a Guice managed object prepares every required object through dependency injection.

## FlightSearch application with Struts 2

Let us examine the artifacts one by one, which illustrate how quickly we could retrofit our application to use Struts 2 with Guice 3.

## FlightServletContextListener

Here in the `configureServlets(...)` API, we simply need to install `Struts2GuicePluginModule` before installing our `MainModule`. Next, to it we need to declare a filter mapping indicating all the requests need to pass through `StrutsPrepareAndExecuteFilter` and provide it a singleton scope. Following listing shows the required piece of code.

```
install(new Struts2GuicePluginModule());

bind(StrutsPrepareAndExecuteFilter.class).in(Singleton.class);
    filter("/*").through(StrutsPrepareAndExecuteFilter.class);
```

## Index page

The index page uses struts tags for rendering form and processing the request. The form variables on the index page appear as the properties of the `FlightSearchAction` class. Using `ValueStack` and `OGNL`, these are directly bound. This relieves us of the coupling between framework components and application logic. The relevant piece of code is as follows:

```
<%@ taglib prefix="s" uri="/struts-tags" %>
<s:form action="Search">
    <s:textfield name="source" label="From"/>
    <s:textfield name="destination" label="To"/>
    <s:textfield name="date" label="Date"/>
    <s:submit label="Submit"/>
</s:form>
```

## FlightSearchAction

FlightSearchAction is annotated with @Namespace and @ResultPath annotation to indicate the routing path. These are used while the action classes are being scanned by, for the declaration of mappings between URL and action mapping while the application is being loaded.

```
@Namespace("/")
@ResultPath(value="/")
public class FlightSearchAction extends ActionSupport{...}
```

Similarly, execute API of FlightSearchAction is also annotated with @Action annotation, which has a value parameter indicating a complete URL to be mapped against a executable API of Action class. This URL is the action of the search form in the index.jsp page.

Execute API of FlightSearchAction handles one complete instance of a work. Here it is to process the request and to return the SUCCESS signal. In case of any checked exception handling or any adverse condition, a different value could be returned, which could be any of the values from the com.opensymphony.xwork2.Action interface. The following code snippet shows the relevant piece of code where execute API of the FlightSearchAction is annotated with the @Action annotation, illustrating various result mappings.

```
@Action(value="Search", results={
    @Result(name="success", location="response.jsp"),
    @Result(name="error", location="error.jsp")
},
exceptionMappings={
    @ExceptionMapping(
        exception="org.packt.exceptions.NoCriteriaMatchException",
        result="error",
        params={"message", "No match found for the supplied criteria"}),
    @ExceptionMapping(
        exception="org.packt.exceptions.NoFlightAvailableException",
        result="error",
        params={"message", "No Flight available exception"})
})
public String execute() {...}
```

Here, we could see that for various results, we have mapped different JSPs. We have mapped response.jsp to success and error.jsp to error. Also we could provide mappings for the run-time exception, if any, which result during execution of the request.

For instance, we throw two different kinds of run-time exceptions from the `FlightEngine.process`; these are `NoCriteriaMatchException` and `NoFlightAvailableException`. We could even provide the mapping for these and redirect them to the appropriate result location:

```
@Inject
private RequestCounter requestCounter;

@Inject
private FlightEngine flightEngine;

@Inject
private Provider<SearchRequest> provider;
```

The singleton scoped, request scoped, and session scoped dependencies are injected and are used in a more simpler fashion than during pure web development.

## Response page

The response page uses struts tags to display the properties of the `SearchResponse` object. The list of `SearchResponse`, which is the result of invoking `FlightEngine.processRequest(...)` is now assigned to a class attribute. This makes it directly accessible in the JSP page as it is present on the `ValueStack` and accessible by `OGNL`. The relevant piece of code is as follows:

```
<%@ taglib uri="http://struts.apache.org/tags-bean" prefix="bean"%>
<%@ taglib uri="http://struts.apache.org/tags-logic" prefix="logic"%>

<logic:iterate name="results" id="searchResponse">
  <bean:write name="searchResponse" property="flightNumber"/>
</logic:iterate>
```

## Summary

This chapter focused on integrating Guice 3 with Struts 2. Installing the module `Struts2GuicePluginModule` in our `FlightServletContextListener.configureServlet(...)` API and registering `StrutsPrepareAndExecuteFilter` to map `/*` all the URLs to it are the only steps we need to take, to prepare a Struts 2 application to use Guice 3 dependency injection features. In the next chapter we are going to implement persistence in our application using JPA and Hibernate using Guice.



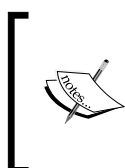


# 6

## Integrating Guice with JPA 2

This chapter illustrates integration of Guice with JPA 2 and Hibernate 3 in a web and stand-alone application. A familiarity with the JPA and Hibernate is desired, particularly in a web environment. However, we will begin with a gentle introduction to JPA and Hibernate. Post that, we will continue with our Struts 2 retrofitted application from the last chapter and make it driven by a JPA, Hibernate-backed database. In this process, we will see how to fit Guice in both standalone as well as a web application to be used as a persistence provider.

We will introduce the concept of transaction in a stand-alone application and in a web application with two strategies like Session per transaction and Session per request.



The guice-persist is the extension for working with JPA and Hibernate. Along with this, there are a couple of dependencies we need, like hibernate-entitymanager, hibernate-annotations, and hibernate-jpa-2.0. These are provided as maven dependencies and are available in packaged WAR.

We would stick to JPA 2.0 standard while using Hibernate 3 as a vendor implementation. For a database, we will use MySQL, although any database could be chosen for this purpose. The reader needs to change the relevant driver classes and configuration properties to use a different database. Please consult the Appendix for installing MySQL on your machine and also how to dump the data we will need to run the application.

## Introduction to JPA 2 and Hibernate 3

**Java Persistence API (JPA)** is a specification for accessing, persisting, and managing data between Java objects or classes and a relational database. JPA is considered to be a standard approach for **Object to Relational mapping (ORM)** in Java.

JPA 2 is a work of the JSR 317 Expert Group. JPA 2 in itself is not a product, but is merely a set of interfaces which define how the persistence mechanism should work. There are a couple of open source and proprietary implementations of JPA 2 available. Hibernate, EclipseLink, and OpenJPA are prominent among them. We will be using Hibernate 3 as an implementation of the JPA 2 specification.

Hibernate is an object-relational-mapping library for java language, providing a framework for mapping an object-oriented-domain model to a relational database. Hibernate solves the problem of object-relational-impedance mismatch by replacing direct persistence-related database accesses with high-level object handling functions. Hibernate 3.5.0 and up is a certified implementation of Java Persistence API 2.0 specification via a wrapper for the core module, which provides conformity with the JSR 317.



For more information related to Hibernate and JPA, readers are encouraged to have a look at the following links:

Hibernate: [http://en.wikipedia.org/wiki/Hibernate\\_\(Java\)](http://en.wikipedia.org/wiki/Hibernate_(Java)) and JPA:  
[http://en.wikipedia.org/wiki/Java\\_Persistence\\_API](http://en.wikipedia.org/wiki/Java_Persistence_API)

JPA 2 specifies various artifacts and interfaces, which help us to complete object relational mapping between various POJOs and database tables, and subsequently executing various operations over them. Let's glance over some important interfaces and artifacts.

## Persistence Unit

A persistence unit represents the set of all classes that are grouped or related in an application. In simple terms, these are the POJOs, which are mapped to tables in a database.

## PersistenceContext

`PersistenceContext` represents a set of entity instances in which for any persistent entity, there is a unique entity instance. In `PersistenceContext`, the entity instances and its lifecycle is managed by an Entity manager. `PersistenceContext` could be scoped either to a transaction or an extended unit of work.

For preparing a persistence context, the declarative way is the usual approach. The declaration for a persistence context is provided in a `persistence.xml` file, which resides in classpath in `META-INF` folder. This file contains definition about a particular persistence unit, wherein what classes are part of it. It also contains other properties, such as a database connection, and a database dialect.

## EntityManagerFactory

An `EntityManagerFactory` creates instances of `EntityManager` for a particular `PersistenceContext`. It could only create instances of an `EntityManager` for a particular `PersistenceContext`. It is equivalent to `SessionFactory` in Hibernate.

## EntityManager

`EntityManager` is used to access the database in a particular unit of work. It is used to create, remove, and query persistent entity instances. This is equivalent to session in Hibernate. Although read-only operations could be completed outside the purview of a transaction, insert, update, and delete operations need to be covered under transactions. Generally, transactions are enabled declaratively using annotations, and are performed using interceptors.

## Session Strategies

There are two types of session strategies generally considered for different situations.

- **Session-per-transaction:** In this, a session (in our case, `EntityManager`) is prepared just before a transaction is required, and closed thereafter. This is not a preferred strategy, but is mostly used in standalone applications and unit testing.
- **Session-per-request:** In this, a session is prepared once a request is received. For example, in a web application, the session is opened once a request is received. This is usually accomplished using a `Filter`. Generally, these are `OpenSessionInViewFilter` for Hibernate or `OpenEntityManagerInViewFilter` for JPA specification. The session is closed once the filter finishes the work.

## Guice 3 with JPA 2

The extension `guice-persist` provides various classes, which facilitate the injection of `EntityManager` in our data access layer classes.

## PersistService

PersistService is an interface, exposing two APIs, start and stop. The implementations of these two APIs means preparing and destroying the EntityManagerFactory instance, using Persistence.createEntityManagerFactory() API.

## UnitOfWork

UnitOfWork is an interface, exposing two APIs, begin() and end(). The implementations of these two APIs necessarily mean creating an EntityManager instance using the EntityManagerFactory created earlier. It serves an essential purpose, because it opens the session for read-only operations, and closes them accordingly.

## JpaPersistService

It is the implementation of the two interfaces we discussed as well as the Provider for EntityManager. JpaPersistService takes a String and a Properties reference as an argument to its constructor. JpaPersistService prepares and destroys the EntityManager when called to start() and stop() arrives respectively. On the other hand, begin() and end() APIs begin and end a transaction using the EntityManager prepared earlier. Through the get() API, the provider provides the reference to the EntityManager prepared.

## PersistFilter

This filter basically acts to start or stop the persistence service while the request is being made. It implements the Session-per-request strategy, a common strategy in web applications. During the initialization of the filter, the start() method is invoked, while during the destruction of the filter, the stop() method is invoked. In doFilter() API while handling the request, the request-handling portion is wrapped in invocation calls of begin() and end(). This basically makes the EntityManager available for database read-only operations.

## @Transactional

Transaction is a cross-cutting concern. There is a little point in managing transactions on our own, particularly in data access classes. It makes a lot of sense to get these accomplished using framework utility classes. Guice provides `@Transactional` interface, which essentially means to include the annotated method inside a transaction. All the methods annotated with `@Transactional` are intercepted and wrapped around calls to begin and conclude the transaction. In the event of an exception, necessary steps to rollback or ignore the exception could also be configured via this annotation. The important caution is that whatever methods which are being annotated with this annotation, must be publicly accessible and should be present in classes which are managed by Guice. The invocation of `@Transactional` annotated methods should be done only on Guice managed objects, which are effectively injected instance variables.

For example, the `getResults()` method of `FlightJPASupplier` is annotated with `@Transactional`. This method is invoked on the instance of `FlightJPASupplier`, which is injected in `FlightEngine`.

## JpaLocalTxnInterceptor

Before we proceed to discuss `JpaLocalTxnInterceptor`, let's look at what is an `Interceptor`? An `Interceptor`, or more precisely a method, intercepts an invocation call to a matching method and wraps the original invocation call inside its own execution. This methodology is suitable to handle cross-cutting concerns. Such a way to handle cross-cutting concerns falls under the purview of Aspect Oriented Programming (AOP). We discuss AOP implementation of Guice in *Chapter 8, AOP with Guice*.

The interceptor whose `invoke()` comes into the picture once the methods annotated with `@Transactional` are invoked, accomplishes following things:

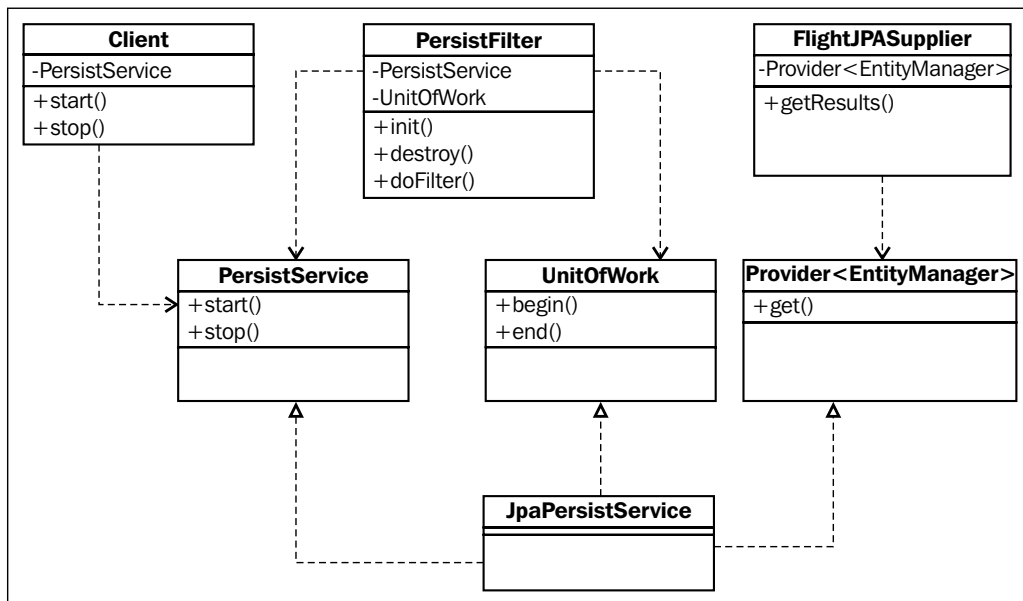
- Checks if a unit of work has started; if not, start it.
- Begins a transaction.
- Allows normal method invocation to proceed.
- Ends a transaction.

- In case of an exception which is configured for rolling back a transaction, the transaction is rolled back.
- In case of an exception which is configured for ignoring, the transaction is committed, and an exception is propagated.
- Essentially, it ends a unit of work.
- Returns the result.

## JpaPersistModule

This module basically configures the `JpaPersistService` and binds it to the interfaces `PersistService` and `UnitOfWork` as the implementation. The constructor argument is the name of the persistence unit configured in the `persistence.xml` file. Properties are derived from the `persistence.xml` file itself.

The following figure shows various artifacts and their relationship with each other:



## FlightSearch application with JPA 2 and Hibernate 3

We will retrofit the application to run in two different modes. First in a standalone mode, we will use the Client as our pivot. Later, we will refactor and use it for a web application.

### Persistence.xml

Persistence.xml contains the declaration for a persistence-unit. The following code shows that:

```
<persistence-unit name="flight" transaction-type="RESOURCE_LOCAL">
  <provider>org.hibernate.ejb.HibernatePersistence</provider>
  <class>org.packt.supplier.SearchResponse</class>
  <properties>
    <property name="hibernate.dialect"
      value="org.hibernate.dialect.MySQL5InnoDBDialect"/>
    <property name="hibernate.connection.driver_class"
      value="com.mysql.jdbc.Driver"/>
    <property name="hibernate.connection.driver"
      value="com.mysql.jdbc.Driver"/>
    <property name="hibernate.connection.url"
      value="jdbc:mysql://localhost/flights"/>
    <property name="hibernate.connection.user" value="flight"/>
    <property name="hibernate.connection.password"
      value="flight"/>
  </properties>
</persistence-unit>
```

Persistence-unit basically defines what needs to be encompassed in an instance of `PersistenceContext`.

### Client

Client has two extra APIs to encapsulate service initialization and destruction. In standalone mode we need to invoke the `service.start()` and `service.stop()` methods manually. The following code shows that:

```
public static void main(String args[]){
  Injector injector = Guice.createInjector(
    newJpaPersistModule("flight"), new MainModule());
```



```
        Client client = injector.getInstance(Client.class);
        client.startService();
        client.makeRequest();
        client.stopService();
    }
```

A notable point here is Session per transaction strategy. Before we make a transaction, we prepare the EntityManagerFactory using `startService()`, and later destroy it using `stopService()`. The `unitOfWork.begin()` and `unitOfWork.end()` invocation happens inside the `JpaLocalTxnInterceptor.invoke()` method.

We need to inject the `PersistService` inside the `Client`, and use it from encapsulated methods. The following code explains how:

```
@Inject
private PersistService persistService;

private void startService() {
    persistService.start();
}

private void stopService() {
    persistService.stop();
}
```

## SearchResponse

`SearchResponse` is annotated with `@Entity`, and hence is an entity in the persistent unit `flight`. Its instances are available in a persistence context, which is looked by an instance of an `EntityManager`. The following code shows that:

```
@Entity
@NamedQueries({
    @NamedQuery(name="SearchResponse.findAll", query="select s from SearchResponse s")
})
@Table(name="flight")
public class SearchResponse implements Comparable<SearchResponse>,
    Serializable{
    private static final long serialVersionUID = 1L;
    @Id
    private Long id;
}
```

There are named queries, which we will be using from our data access layer. Let's have a look at our data access layer.

## FlightEngineModule

Declaration to bind an instance of the `FlightSupplier` interface with an implementation of `FlightJPASupplier` is provided in the `FlightEngine` module. Let's look at it in the following code:

```
bind(FlightSupplier.class)
    .annotatedWith(Names.named("jpa"))
    .to(FlightJPASupplier.class).in(Singleton.class);
```

## FlightJPASupplier

`FlightJPASupplier`, like other suppliers, implements the `FlightSupplier` interface and is injected in `FlightEngine` using `@Named` annotation. Following is a relevant piece of code from `FlightEngine`:

```
@Inject
@Named("jpa")
private FlightSupplier flightJPASupplier;
```

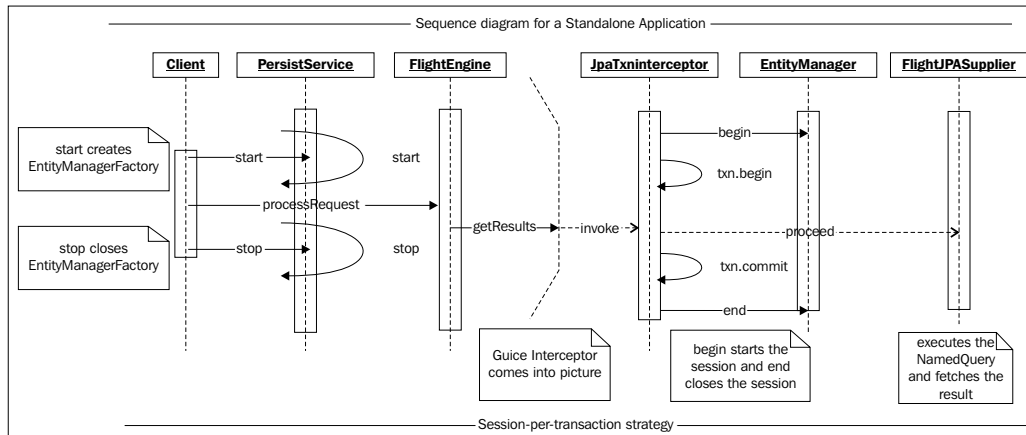
We will be using `flightJPASupplier` to retrieve the results rather than other suppliers. Let's look at the `FlightJPASupplier.getResults()` in the following code:

```
@Inject
private Provider<EntityManager> entityManagerProvider;

@Override
@Transactional
public Set<SearchResponse> getResults() {
    Query query =
        entityManagerProvider.get().createNamedQuery("SearchResponse.
            findAll");
    Set<SearchResponse> resultSet = new HashSet<SearchResponse>();
    resultSet.addAll((List<SearchResponse>) query.getResultList());
    return resultSet;
}
```

The method is annotated with `@Transactional`, and hence provides us declaratively managed transactions. Also, the `EntityManager` is provided by the `entityManagerProvider`, which is provided by `JpaPersistenceService`. We query in a persistence context using the named queries, and fetch the result.

The following sequence diagram highlights how the call propagation progress in stand-alone mode. This was the example of Session-per-transaction strategy.



SearchRequest, annotated with @RequestScoped, could be a problem. Because we are operating out of a web context, the injector initialization would fail. We need to remove the @RequestScope before attempting to build the solution for the stand-alone mode. Once we attempt to deploy the solution as a web application, we need to apply the @RequestScoped, and this should work normally.

Now, let's look at what changes are required for making the web application work.

## FlightServletContextListener

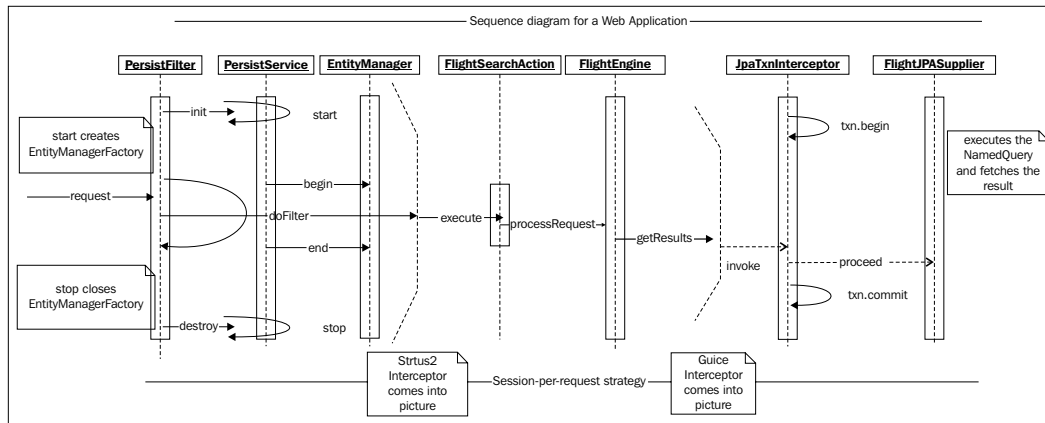
FlightServletContextListener needs to configure an instance of JpaPersistModule. Afterward, we configure an instance of Struts2GuicePluginModule, and then our MainModule. Later on we just need to configure PersistFilter to provide us OpenEntityManagerInView available for readOnly queries.

```

protected void configureServlets() {
    install(new JpaPersistModule("flight"));
    install(new Struts2GuicePluginModule());
    install(new MainModule());
    bind(StrutsPrepareAndExecuteFilter.class).in(Singleton.class);
    filter("/*").through(StrutsPrepareAndExecuteFilter.class);
    filter("/*").through(PersistFilter.class);
}

```

The following sequence diagram makes it clear how the request propagation works in a web application. This was the example of Session-per-Request strategy.



## Summary

We have learned to use the artifacts provided by the guice-persist extension to integrate JPA, and Hibernate within a web application powered by Struts 2, and also in a stand-alone application. We also implemented various session strategies in different scenarios. Also, we saw that how simple and non-conflicting it is to start a unit of work from a Guice-injected class, and fetch the record from the database. In next chapter, we will discuss advanced areas of Guice where we will discuss how Guice could be extended to develop plugins and extensions. This discussion will be carried out through Guice **Service Provider Interfaces (SPI)**.



# 7

## Developing Plugins and Extensions using Guice

In this chapter, we will visit important classes and interfaces which are useful for developing extensions and plugins around Guice. We will also explore these interfaces in terms of analyzing various kinds of bindings in **guice-multibinder**.

In this process, we will separate out **MainModule** from our application, and make it a part of a custom plugin that we are developing. This custom plugin would contain various examples related to SPI. This custom plugin is located at `chapter_7/guice-analyst` as a maven project; and the sample project, which uses this plugin is located at `chapter_7/flights-struts-jpa-ext`.

### Developing a plugin

A Guice plugin is simply a JAR file which separates commonly used functionalities. Developing a separate plugin could provide us a variety of benefits. Few of these could be like grouping various initialization tasks, preparing factories to instantiate objects, and so on. A separate plugin could also be used to develop interfaces and classes, which could help us to analyze bindings and write custom extensions.

In our case, we find it a good option to separate functionality of initializing and loading our custom modules. Separating the functions, which are not dependent on application logic increases reusability. We will separate **MainModule**, which loads other modules and make it a part of plugin, `guice-analyst`. This plugin then could be used in any different project; in our case, it is used as **flights-struts-jpa-ext**.



The Guice plugin dependency is available as the `guice-analyst-1.0-SNAPSHOT.jar`. This dependency is included in the `pom.xml` for `chapter_7/flights-struts-jpa-ext`. For building up the plugin we need to execute the following command, in `chapter_7/guice-analyst` directory:

```
$shell> mvn clean install
```

Build the plugin and install it in the local maven repository using the preceding command.

We will rename it from `MainModule` to something more sensible, **`AssistInstallModule`**. We will be adding an extra argument to the default constructor of `AssistInstallModule`, which will help us to figure out which package to refer when looking up for the `Module` classes.

```
private String packageName;
public AssistInstallModule(String packageName) {
    this.packageName = packageName;
}
```

Next, there are minor changes to load classes from the provided package name:

```
ClassPath classPath =
    ClassPath.from(AssistInstallModule.class.getClassLoader());
for( ClassInfo classInfo:
    classPath.getTopLevelClassesRecursive(packageName)) {
}
```

The plugin could then be used for loading all the modules in a particular package. Here is the snippet from the `Client.java` file:

```
Injector injector = Guice.createInjector(
    new JpaPersistModule("flight"),
    new AssistInstallModule("org.packt.modules")
);
```

The `AssistInstallModule` is now available to be used independently.

## Guice SPI (Service Provider Interface)

**Service Provider Interface (SPI)** is a mechanism used in the development of Java class libraries and standard extension to the language itself. It exhibits certain interesting properties, which could lead to the development of reusable components.

**From official documentation**

"A service is a well-known set of interfaces and (usually abstract) classes. A service provider is a specific implementation of a service. The classes in a provider typically implement the interfaces and subclass the classes defined in the service itself. Service providers can be installed in an implementation of the Java platform in the form of extensions, that is, JAR files placed into any of the usual extension directories. Providers can also be made available by adding them to the application's class path or by some other platform-specific means."

SPIs are implemented using interfaces and abstract classes. An interface contains declarations for constants and methods. Interfaces are implemented by classes, which would become part of service providers. An essential feature of SPI is that variables are declared using interface data type. These variables can then provide reference to any object implementing that interface, and any methods declared in the interface could be accessed.

Using interfaces in this way allows the client application to invoke methods on an object whose implementing class is not known at compile time. At runtime the Java class-loading mechanism is used to locate and load classes that implement SPI dynamically.

There are many well-known uses of SPIs. Prominent among them are JDBC, JCE, JNDI, and JAXP.

Guice SPI or the Guice Service Provider Interface is a package, which assists in developing tools and extensions around Guice. It could be divided into two parts:

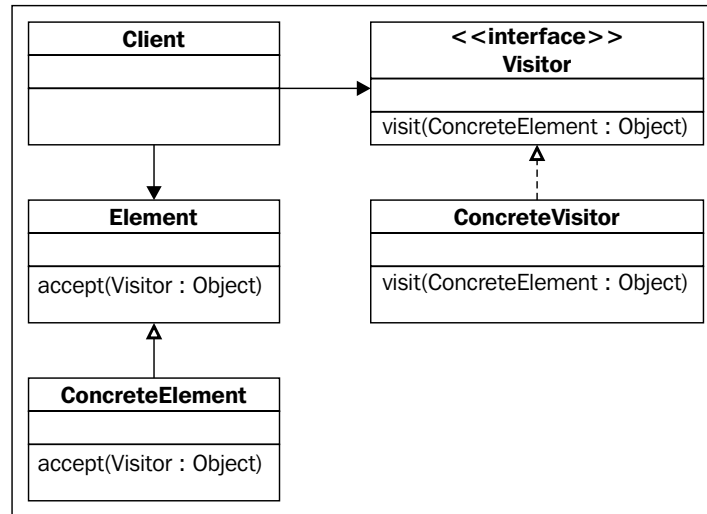
- **Elements SPI**, which provides a way to inspect module bindings.
- **Extensions SPI**, which provides a way of interaction between Guice core and extensions. It primarily serves to develop extensions and to inspect injector bindings.

## Quick introduction to Visitor Design pattern

Guice SPIs are developed using **Visitor Design** pattern. A Visitor Design pattern is a behavioral design pattern, which represents an operation to be performed on the elements of an object structure. A visitor lets you define a new operation without changing the classes of elements on which it operates.



Before visiting any of the SPI packages from Guice, we would be taking a whirlwind tour of Visitor Design pattern.



Assuming there are two objects types, one `Element` and other a `Visitor`. An `Element` object has an `accept()` method, which can accept arguments of type `Visitor`. The `accept()` method calls a `visit()` method of `Visitor`, the `Element` object passes itself inside the `visit()` method as an argument.

Thus, on invocation of the `accept()` method, its implementation is chosen on both:

- Dynamic type of the `Element` object type
- Static type of the `Visitor` object type


Subsequently, on invocation of the `visit()` method, its implementation is chosen on both:

- Dynamic type of `Visitor`
- Static type of `Element`

Consequently, implementation of the `visit()` method is always chosen based on both:

- Dynamic type of `Element`
- Dynamic type of `Visitor`

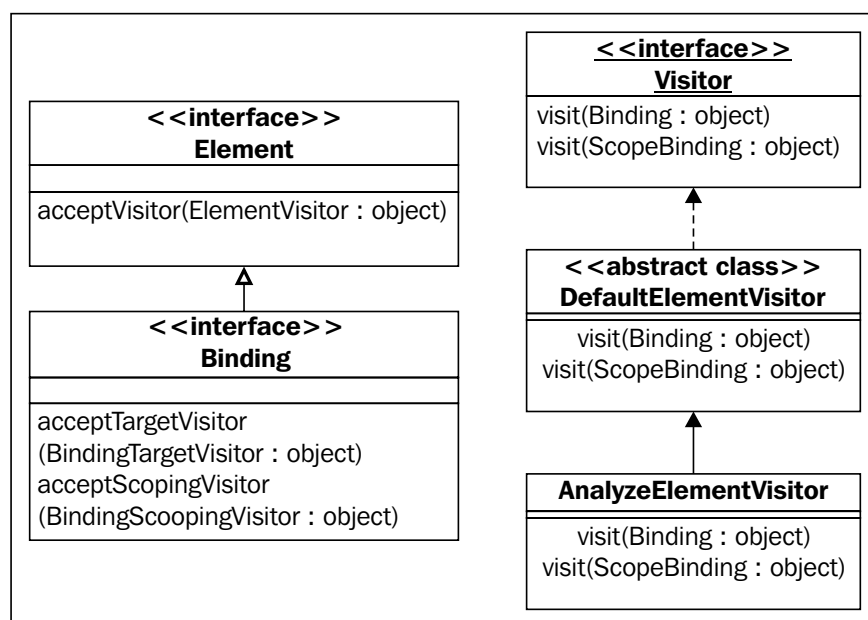
In this way, different kinds of visitors could interact with different kinds of elements, based on their dynamic types.


 Readers interested in learning more about the Visitor Design pattern are suggested to visit [http://en.wikipedia.org/wiki/Visitor\\_pattern](http://en.wikipedia.org/wiki/Visitor_pattern).

## Elements SPI

Elements SPI provides a way to inspect the elements before injection happens. We will have a look at the important interfaces and classes.

The **Element** and **ElementVisitor** interfaces represent an "element" and a "visitor" (analogy in Visitor Design pattern). The Element interface is a core component of a module, and the ElementVisitor interface visits the elements. The **DefaultElementVisitor** interface is an abstract implementation of the ElementVisitor interface. We subclass the DefaultElementVisitor in **AnalyzeElementVisitor** interface and override a few methods to analyze the visitor behavior.



## Implementing Elements SPI

Let's pick an implementation of Element to analyze its behavior. The Element interface is extended by Binding. Binding is a mapping from a key to the strategy for getting instances of the type. Let's look at the following lines of overridden method in AnalyzeElementVisitor. Here, we visit the bindings, which are of type FlightSupplier:

```
@Override
public <FlightSupplier> Void visit(Binding<FlightSupplier> binding) {
    Key<String> key = binding.getKey();
    System.out.println("Key :" + key.getTypeLiteral());
    System.out.println("Annotation : " + key.getAnnotation());
    System.out.println("Source : " + binding.getSource());
    return visitOther(binding);
}
```

Here, we have overridden the visit(Binding<V> binding) method from the DefaultElementVisitor. Now, we would visit the various elements, which belong to different modules. This happens at the AssistedInstallModule's configure() method.

```
AnalyzeElementVisitor defaultElementVisitor =
    new AnalyzeElementVisitor();

for(Element element :
    Elements.getElements(Stage.DEVELOPMENT,modules)) {
    element.acceptVisitor(defaultElementVisitor);
}
```

Here, the element accepts the visitor and later on when visitor.visit() is called AnalyzeElementVisitor's API come into picture. When we run the application in the standalone mode using command, the following binding information is printed:

```
$shell>mvn exec:java -Dexec.mainClass="org.packkt.client.Client"
```

```
Key :org.packkt.supplier.FlightSupplier
Annotation : @com.google.inject.name.Named(value=xmlSupplier)
Source : org.packkt.modules.FlightEngineModule.configure(
FlightEngineModule.java:26)

Key :org.packkt.supplier.FlightSupplier
```

```
Annotation : @com.google.inject.name.Named(value=jpa)
Source : org.packt.modules.FlightEngineModule.configure(
FlightEngineModule.java:34)
```

```
Key : java.lang.String
Annotation : @com.google.inject.name.Named(value=dateFormat)
Source : org.packt.modules.FlightUtilityModule.configure(
FlightUtilityModule.java:11)
```

Following invocation of `visitor.visit()` shows the explicit bindings we have applied at the various modules. The information is revealed about a particular binding as for what key a mapping exists. For example, in this case for the key "FlightSupplier", there are two annotations. Element extension not only provides the place to analyze the various bindings, but also to manipulate them if required. The bindings we analyzed are the Module bindings; and the bindings, which are available from Injector, are the injector bindings.



**Module Bindings** are retrieved through the elements SPI. Such bindings are incomplete and inconclusive. As a consequence, they could not be used to provide instances. We have introspected the bindings from modules; at the moment, the injector isn't completely ready, because the Object graph to complete the injector is not ready. For example, the preceding binding inspection reveals information about the key, annotation, and source, but doesn't conclusively provide information about how the object would be instantiated and injected.

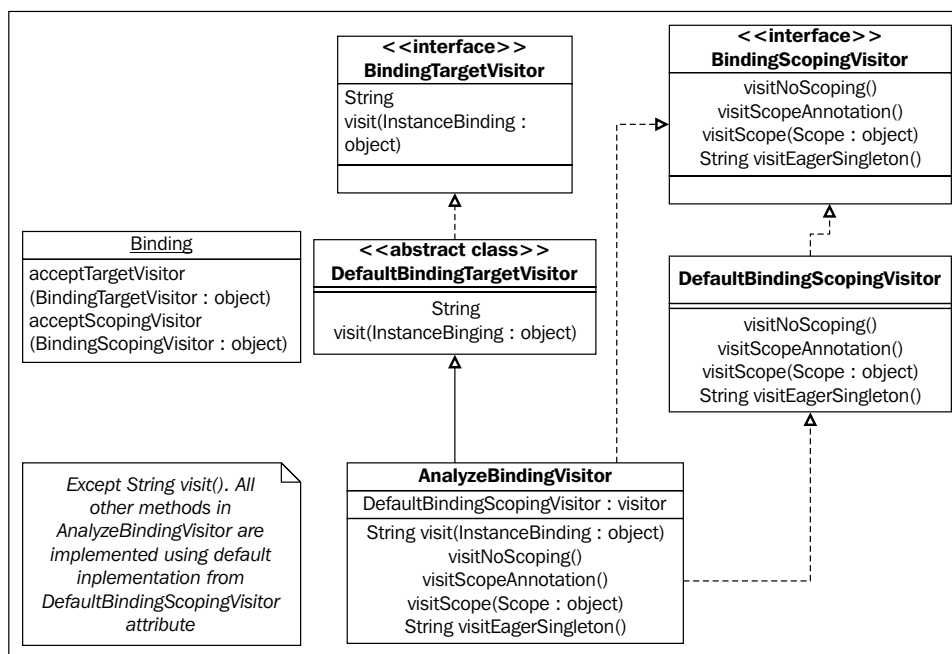
**Injector Bindings**, on the other hand, are complete. They can be used to create instances. A major difference comes from the fact that during their inspection, the Injector is ready to fulfill dependency injection.

## Uses of Elements SPI

Elements SPI implementation is a static analysis of the various module bindings when they are not complete. This could be used for various purposes, such as testing types bound to an implementation without even creating the Injector. Such a facility is valuable to test the same configuration in different environments. Another use could be to enforce various policies, like inhibiting static injection and logging it as an error.

## Extensions SPI

Extensions SPI serves two purposes; first it provides a convenient way to analyze the Injector bindings, and second it provides artifacts to write our own extensions. First, we will have a look at the inspection of the injector bindings. Let's look at the following diagram:



Over here, we are exploring the Binding interface further. Binding, in addition to `acceptVisitor(ElementVisitor visitor)` from the extended Element interface, provides two more APIs to support visiting injector bindings. These are:

- `acceptTargetVisitor(BindingTargetVisitor visitor)`
- `acceptScopingVisitor(BindingScopingVisitor visitor)`

These APIs help us inspect various kinds of completed bindings, which are available from injector only once the complete object graph is available for the injection.

Let's explore the BindingTargetVisitor first. In our custom implementation of BindingTargetVisitor, we extend the DefaultBindingTargetVisitor interface and override the APIs we need to inspect.

```
public class AnalyzeBindingVisitor extends
    DefaultBindingTargetVisitor<Object, String> implements
```

```

BindingScopingVisitor<String>{

    public String visit(InstanceBinding<?> binding){
        Key<?> key = binding.getKey();
        System.out.println("Key :" + key.getTypeLiteral());
        System.out.println("Annotation : " + key.getAnnotation());
        System.out.println("Source : " + binding.getSource());
        System.out.println("Instances :
        "+binding.getInstance().toString());
        return visitOther(binding);
    }
}

```

Here, we could see that `InstanceBinding` (which in fact extends from `Binding` interface) has one more API, which refers to the instance, which needs to be injected. In this way, it is a complete binding, which is in fact a reflection of the `Injector` itself.

As we discussed, these bindings are only available when the complete object graph and `Injector` is prepared. We could only apply it where we have a reference of `Injector` available to us.

Following is the code snippet from the `Client` class where we prepare an instance of `AnalyzeBindingVisitor` and use it to inspect various injector bindings.

```

AnalyzeBindingVisitor analyzeBindingVisitor = new
AnalyzeBindingVisitor();
for(Binding<?> binding : injector.getBindings().values()){
    System.out.println(
        binding.acceptTargetVisitor(analyzeBindingVisitor));
    };
}

```

When we run the application, we can see the following output:

```

Key :org.packt.supplier.FlightSupplier
Annotation : @com.google.inject.multibindings.
Element(setName=,uniqueId=2)
Source : org.packt.modules.FlightEngineModule.
configure(FlightEngineModule.java:39)
Instance : org.packt.supplier.JSONSupplier@717535b6

Key :java.lang.String

```

**Annotation :** `@com.google.inject.name.Named(value=dateFormat)`

**Source :** `org.packt.modules.FlightUtilityModule.  
configure(FlightUtilityModule.java:11)`

**Instance :** `dd-MM-yy`

Here we could see that the binding is complete, because it has different instances available to complete the injection. In a similar way, we could also look at how the Scope injection works.

Let's explore the `acceptScopingVisitor()` method, which accepts an argument of type `BindingScopingVisitor`. `AnalyzeBindingVisitor` implements `BindingScopingVisitor` and complements the default implementation by having a reference to `DefaultScopingVisitor`.



**AnalyzeBindingVisitor** is in fact an Adapter class, which serves as implementation of the `BindingTargetVisitor` as well as `BindingScopingVisitor` interfaces. Here, we extend from the `DefaultBindingTargetVisitor` class because it is abstract, and it has a dependency on `DefaultBindingScopingVisitor` as an attribute. (This also makes sense because it is a non-abstract class.)

Here is one of the implemented APIs:

```
public String visitScope(Scope scope) {  
    return scope.toString();  
}
```

Also, we need to invoke the `acceptScopingVisitor()` method while we are iterating over various bindings in the `Client` class.


```
System.out.println(  
    binding.acceptScopingVisitor(analyzeBindingVisitor));
```

We just print the information related to the scope of a binding. On command line post execution, it would print the information about the various `Scope` instances participating in dependency injection.

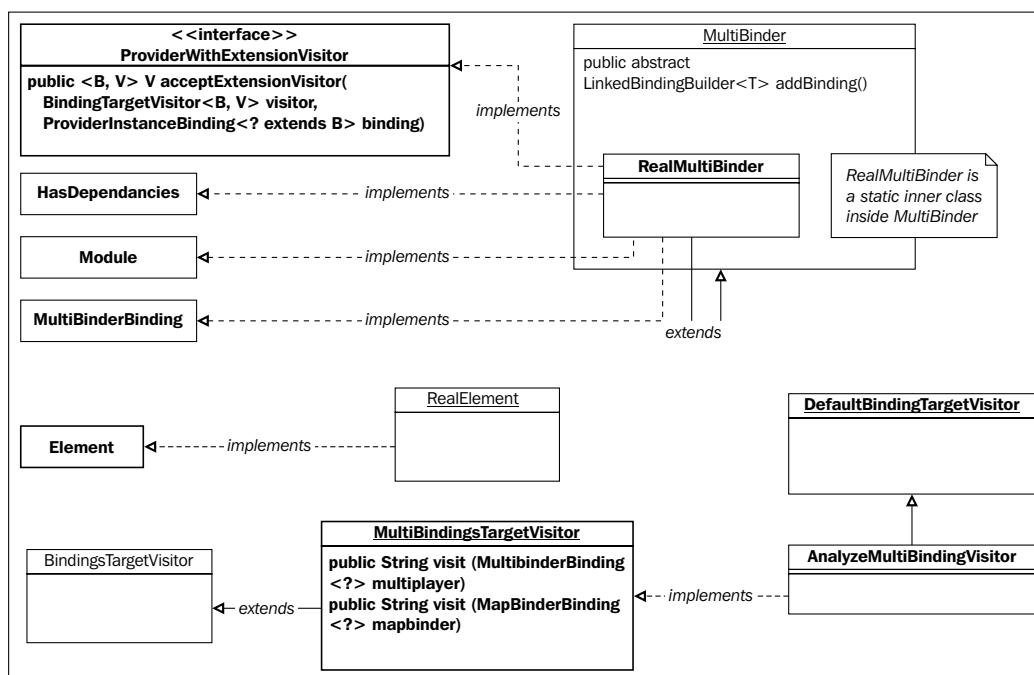
```
org.packt.scope.CSVScope@12b9b67b  
Scopes.SINGLETON
```

## Exploring a custom extension, guice-multibinder

Exploring a custom extension involves understanding how the custom extension works and what purpose it serves. In this topic, we will analyze **guice-multibinder**. During this, we will first inspect the important classes and interfaces in it, and later, how it provides an implementation of the extensions SPI. We will also explore the binding collection case in `FlightEngineModule`, where we prepare a collection of bindings using guice-multibinder.

 We discussed guice-multibinder extension in *Chapter 3, Diving Deeper in Guice* to illustrate binding collections. We pick it up again to explore the extensions SPI, because it gives us a good ground to analyze Injector bindings and a custom extensions case.

Let's look at the following class diagram for a few of the classes and interfaces in guice-multibinder extension. This is not a complete diagram, and discusses only relevant pieces, which come into action while we are discussing its internal working.





## Important classes and interfaces in guice-multibinder

The important classes to be considered here are `Multibinder`, `MultibindingsTargetVisitor`, `RealMultibinder`, and `RealElement`.

- **Multibinder**: It is an abstract class, which exposes overloaded static API `newSetBinder()` with a variety of arguments, a binder, and a `TypeLiteral` or a `Class`, and an `Annotation`. The rationale behind these various overloaded APIs is allowing variety of usage for preparing the key for binding. An example of overloaded API is as follows:

```
public static <T> Multibinder<T>
    newSetBinder(Binder binder, TypeLiteral<T> type)
```

Using such overloaded APIs, `Multibinder` returns a reference to it, which actually is an instance of inner static class `RealMultibinder` (refer to the preceding diagram). The reference to this instance is returned after installing it, because it also implements `Module`.

- **RealMultibinder**: It is an inner static class, which extends `Multibinder` and implements four interfaces, which aid in participation during binding and dependency injection. These interfaces are:
  - **HasDependencies** interface, which is implemented to return any dependencies of this type explicitly.
  - **Module** interface, which is implemented to provide configurations to bind itself as a `Provider` against the `Key` provided.
  - **MultibinderBinding** interface, which represents a binding for a `Multibinder`.
  - **ProviderWithExtensionVisitor** interface, which is implemented to be a `Provider` that is a part of the extension, which supports an implementation of `BindingTargetVisitor`. Typically, here we will place an `instanceOf` check to make sure we are visiting only the implementation of `BindingTargetVisitor` that our custom extension supports. In this case, `BindingTargetVisitor` is further extended by the `MultibindingsTargetVisitor` interface.
- **MultibindingsTargetVisitor** extends `BindingTargetVisitor` and adds its own methods to visit `MultibinderBinding` and `MapbinderBinding`.
- **RealElement** implements the `Element` interface and assists in preparing the `Module` bindings, which are later turned in `Injector` bindings.

## Multibinder in action

A typical usage of **Multibinder** in `FlightEngineModule`'s `configure()` method is:

```
Multibinder<FlightSupplier> multiBinder =
    Multibinder.newSetBinder(binder(), FlightSupplier.class);
```

A reference to the instance of **Multibinder** could then be used to add bindings to the set:

```
multiBinder.addBinding().to(XMLSupplier.class).asEagerSingleton();
multiBinder.addBinding().toInstance(new JSONSupplier());
```

When the `addBinding()` method is invoked a reference to the `LinkedBindingBuilder` interface is returned, which accepts arguments as simple bindings. So we have a mechanism where, for a collection of same type, we could provide variety of bindings for dependency injection. The complete injector bindings could be visited for which the extension developer has to accomplish two things:

- The Binder implementation should implement the `ProviderWithExtensionVisitor` interface to provide an implementation of following method. Let's look at this method in `RealMultibinder`:

```
@SuppressWarnings("unchecked")
public <B, V> V acceptExtensionVisitor(
    BindingTargetVisitor<B, V> visitor,
    ProviderInstanceBinding<? extends B> binding) {
    if (visitor instanceof MultibindingsTargetVisitor) {
        return ((MultibindingsTargetVisitor<Set<T>,
            V>)visitor).visit(this);
    } else {
        return visitor.visit(binding);
    }
}
```

The implementation of this method in `RealMultibinder` involves an `instanceOf` check whether the reference of `BindingTargetVisitor` is actually an instance of `MultibindingsTargetVisitor`. If yes, then it is cast to `MultibindingsTargetVisitor` and its `visit()` method is invoked with `RealMultibinder`'s instance itself. This is because `RealMultibinder` also implements `MultibinderBinding`, which is the elementary interface to support multi binding operations.

If no, then `visit()` method is invoked with `ProviderInstanceBinding` APIs reference.

- In order to visit the Injector bindings, `BindingTargetVisitor` needs to be extended, and its implementation then could be used to visit the Injector bindings. Here, it is extended as `MultibindingsTargetVisitor`, which provides two APIs to visit `MultibinderBinding` and `MapBinderBinding`.

We implement `MultibindingsTargetVisitor` interface in our custom class `AnalyzeMultiBindingVisitor`, which extends from `DefaultBindingTargetVisitor`, hence is a type of `BindingTargetVisitor`.

Following is an implementation of an API from `MultiBindingTargetVisitor`.

```
public String visit(MultibinderBinding<?> multibinder) {

    System.out.println("MultiBinding - Key: " +
        multibinder.getSetKey());
    for(Binding<?> binding : multibinder.getElements()){
        Key<? extends Object> key = binding.getKey();
        System.out.println("Key : " + key.getTypeLiteral());
        System.out.println("Annotation : " +
            key.getAnnotation());
        System.out.println("Source : " +
            binding.getSource());
    }

    return "Key: " + multibinder.getSetKey() + " \n"
        + " uses bindings: " + multibinder.getElements() + " \n"
        + " permitsDuplicates: " +
            multibinder.permitsDuplicates() + "\n";
}
```

Following visitor implementation, when instantiated and supplied as an argument to `acceptTargetVisitor()` while we are iterating over Injector bindings in the `Client` class, following results will be displayed:

```
AnalyzeBindingVisitor analyzeBindingVisitor = new
AnalyzeBindingVisitor()
for(Binding<?> binding : injector.getBindings().values()){
    binding.acceptTargetVisitor(new AnalyzeMultiBindingVisitor());
}

MultiBinding - Key: Key[type=java.util.Set<org.packt.supplier.
FlightSupplier>, annotation=[none]]
```

```
Key :org.packt.supplier.FlightSupplier  
Annotation : @com.google.inject.multibindings.  
Element (setName=,uniqueId=1)  
Source : org.packt.modules.FlightEngineModule.  
configure (FlightEngineModule.java:38)
```

We could clearly see Injector bindings, which are a part of the collection to be injected. Also note that each binding is assigned a unique identifier (sequentially using natural numbers).

## Summary

In this chapter, we explored inner workings of Guice extensions. We went through Guice Service Provider Interface (SPI) and visited Module bindings through Elements SPI and Injector bindings through Extensions SPI. We also learned how custom extensions could be developed, and understood the working mechanism of guice-multi-binder. In the next chapter, we will discuss AOP implementation with Guice.



# 8

## AOP with Guice

This chapter starts with a brief overview of what AOP is and how Guice adds this paradigm to its own style of AOP with method interceptors. We will have a look at the necessary artifacts, such as `Matchers` and `MethodInterceptor`. We will then move over and develop our own `LoggingInterceptor` to intercept and log the details about method invocation.


### What is AOP?

(AOP) **Aspect Oriented Programming** gained significant prominence as a programming methodology for solving problems that are cross-cutting and often throw the same set of challenges across all layers. Such problems do not fall under the purview of Object Oriented Programming. Consider using a logger at each and every method call during the debugging phase or handling run-time exceptions at front layers to provide easy to understand user messages. Another good example would be implementing transaction and security. These cross cutting concerns are handled by the artifacts, which are termed Aspects in AOP.

### How AOP works?

AOP works by separating common concerns across the layers in separate classes and names them *Aspect*. Aspects have APIs which aim to solve a common problem, like logging, and exception handling. These are then invoked while intercepting the actual method invocation. Such APIs are termed Advice. Advices are interceptors, which intercept original method invocation and wrap it around with handling of cross cutting concerns.

Now, for applying Advices to methods, AOP provides artifacts named **JoinPoint**. Using a JoinPoint we could define, which API invocation would be wrapped around by what Advice.



AOP alliance <http://aopalliance.sourceforge.net> is the driving force behind standardization of various AOP implementations. AOP alliance seeks to separate the implementation part from the weaving part. Here, it provides a set of interfaces, which represent the MethodInterception, Advice, and JoinPoint. Guice provides its own implementation for the same.

Weaving is a mechanism by which the aspects are applied to the designated classes and methods. Weaving is generally done by generating byte code for the interceptors, using various libraries for byte code generation libraries, such as JavaAssist, BCEL, and ASM. There are three places, any of which could be used to generate the bytecode. These are while compiling the code (Compile Time Weaving), loading the application known as (Load Time Weaving) and (Runtime Time Weaving) while actual method invocation is running.

## How Guice supports AOP?

Guice conforms to AOP alliance, and so we could directly write interceptors by implementing MethodInterceptor interface (`org.aopalliance.intercept`). APIs could then be annotated with custom annotations, and these annotations could be used to join method invocation with the interceptors. For joining method interception with the interceptor, we need to use the `bindInterceptor(...)` API from the AbstractModule. Let's see all of this in action.

## Implementing a LoggingInterceptor

Consider we need to design a mechanism (a fictitious one), which could be used to log invocation to a method and arguments passed to it. This could be used across the methods. This makes it a good cross-cutting concern. Let's implement this concern as an implementation of MethodInterceptor.

```
public class LoggingInterceptor implements MethodInterceptor{

    @Override
    public Object invoke(MethodInvocation invocation) throws Throwable {
        System.out.println("Invoking"+invocation.getMethod().getName());
        Object[] objectArray = invocation.getArguments();
    }
}
```

---

```

    int i = 0;
    for(Object object : objectArray){
        System.out.println("Argument [" + i
            + "]" + object.getClass().getName());
        ++i;
    }
    return invocation.proceed();
}
}

```

MethodInterceptors invoke API provides MethodInvocation as the argument. MethodInvocation represents the method, which has been intercepted. We could log here the necessary information about the method invocation. Also, we could defer the invocation (depending upon the requirements), and even handle an exception thrown by method invocation.

Next, we need a custom annotation to join intended APIs with this interceptor. Let's develop a custom annotation for the same.

```

@Retention(RetentionPolicy.RUNTIME) @Target(ElementType.METHOD)
public @interface WrapIt {}

```

@WrapIt is the annotation we would be using to join our interceptor with the APIs. Let's annotate two APIs for method interception. These are:

```

@WrapIt
public void makeRequest() {...}

@WrapIt
public List<SearchResponse> processRequest(SearchRequest
    flightSearchRequest) {...}

```

Lastly, we need to bind interceptor with the annotation. This could be easily done in any of the modules. A statement like this would suffice in ClientModule class:

```

bindInterceptor(
    Matchers.any(),
    Matchers.annotatedWith(WrapIt.class),
    new LoggingInterceptor()
);

```

When compiled and run, the following output would be produced:

```

mac-pro25:flights-struts-jpa-aop hussainp$ mvn exec:java -Dexec.
mainClass="org.packt.client.Client"
[INFO] Scanning for projects...

```



```
[INFO] Building flights-struts-jpa-aop 1.0-SNAPSHOT
[INFO] --- exec-maven-plugin:1.2.1:java (default-cli) @ flights-struts-jpa-aop ---
Invoking makeRequest
Invoking processRequest
    Argument [0]org.packt.client.SearchRequest
MAD - AMS
[INFO] -----
[INFO] BUILD SUCCESS
[INFO] -----
[INFO] Total time: 3.028s
[INFO] Finished at: Mon Jun 17 08:32:26 GMT+05:30 2013
[INFO] Final Memory: 8M/81M
[INFO] -----
---
```

The methods were intercepted, and various logging statements printed information about the method invocation and arguments with which these were invoked. Let's look at how the things proceeded. First, we will look into the signature for `bindInterceptor`.

```
protected void bindInterceptor(
    Matcher<? super Class<?>> classMatcher,
    Matcher<? super Method> methodMatcher,
    org.aopalliance.intercept.MethodInterceptor...
    interceptors)
```

The first argument is a type safe (typed to `Class`) implementation of the `Matcher` interface, which indicates, what classes should be targeted for binding the interceptor. The second argument is again a type-safe implementation (typed to `Method`) of `Matcher` interface, indicating what methods of these classes should be targeted for the binding. The last argument is a varargs, which accepts any interceptor instances, which need to be bound to these Matchers.

## Understanding Matchers

Let's look at how the Matchers are prepared. Matchers, as a class provide various convenient static APIs (making it a factory), which let us prepare the convenient matchers for binding. These are as follows:

- `Matchers.any()`: This returns `Matcher<Object> any`, which simply indicates to bind the interceptors to any instance of `Object`. Literally, it means look for every class's instance while scanning. For example, the following usage of `Matchers.any()` would bind each and every public method of the each class to the `LoggingInterceptor`.

```
bindInterceptor(  
    Matchers.any(),  
    Matchers.any(),  
    new LoggingInterceptor()  
);
```

- `Matchers.identicalTo(Object value)`: This returns a `Matcher<Object>` which is same as the object supplied. This is implemented by applying `==` reference check to ensure the object equality. This is a very specific case, because it helps us to bind an object to its interceptor while declaring bindings. Consider an instance which is intended to be provided as an argument to the `.toInstance(...)` method of a binding. The same object instance could be targeted for interception using this API. The following code snippet would bind any methods of the instance of `JsonSupplier` to be intercepted.

```
bindInterceptor(  
    Matchers.identicalTo(jsonSupplier),  
    Matchers.any(),  
    new LoggingInterceptor()  
);
```

- `Matchers.only(Object value)`, returns a `Matched<Object>`, which equals the value of the object supplied. This is implemented by applying `.equals()` check over the object supplied. Hence, it presents a case to intercept each and every object which is equal in value to the object instance provided as an argument. The following binding would bind any instance of `SearchRequest` which equals in value to the object referred by `SearchRequest` to the `LoggingInterceptor`. All the public methods of those instances would be intercepted.

```
bindInterceptor(  
    Matchers.only(SearchRequest),  
    Matchers.any(),  
    new LoggingInterceptor());
```

- `Matchers.inPackage(Package targetPackage)`: This returns an instance to `Matcher<Class>`, which includes all classes within the package. For instance, if we need to bind all the providers to logging interceptor, the following statement would suffice.

```
bindInterceptor(  
    Matchers.inPackage(  
        Package.getPackage("org.packt.supplier.provider")  
    ),  
    Matchers.any(),  
    new LoggingInterceptor()  
);
```

- `Matchers.subPackage(String packageName)`: This returns an instance to `Matcher<Class>`, which includes not only the classes in the given package but also in the subpackage. The following call would bind all the classes and subclasses in the package `org.packt.supplier.provider`:

```
bindInterceptor(  
    Matchers.inPackage("org.packt.supplier.provider"),  
    Matchers.any(),  
    new LoggingInterceptor()  
);
```

- `Matchers.subClassesOf(Class<?> superclass)`: This returns an instance of type `Matcher<Class>` which includes all instances of type superclass and instances of its subclass as well. The following declaration would match any public methods of `CSVSupplier` class instances and instances of its subclasses.

```
bindInterceptor(  
    Matchers.subClassesOf(CSVSupplier.class),  
    Matchers.any(),  
    new LoggingInterceptor()  
);
```

- `Matchers.returns(Matcher<? super Class<?>> returnType)`: This returns an instance of `Matcher<Method>` which is ideally for the second argument of the `bindInterceptor(...)`. For binding every method with a return type of `List`, we would use the following statement:

```
bindInterceptor(  
    Matchers.any(),  
    Matchers.returns(Matchers.subClassesOf(List.class)),  
    new LoggingInterceptor()  
);
```

- `Matchers.annotatedWith(...)`: This returns an instance of `Matcher<AnnotatedElement>`, hence, offering the most convenient way to join an interceptor with the intended API. It is an overloaded API, offering two choices argument of type `Annotation` or argument of type `Annotation.class`. Here in our example, we had prepared the `@WrapIt` annotation to indicate the methods to be intercepted. Simply annotating the methods with the `@WrapIt` and exclusively binding the interceptor with this matcher gets this done.
- `Matchers.not(Matcher<? super T>)`: This interestingly could accept a type of `Matcher<Class>`, `Matcher<Object>`, or `Matcher<Method>` and could invert the matching criteria. Hence, offering a convenient way to negation.

## Limitations

Guice uses runtime bytecode generation for AOP implementation. The newly generated bytecode has invocation calls to Aspects or Interceptors, woven around the intended APIs. This approach has certain limitations, which must be kept in mind while designing the Interceptors and applying Matches to them. These are listed at <https://code.google.com/p/google-guice/wiki/AOP>. These are copied here for the sake of reference.

- Classes must be public or package-private
- Classes must be non-final
- Methods must be public, package-private, or protected
- Methods must be non-final
- Instances must be created by Guice by an `@Inject`-annotated or no-argument constructor
- Method interception doesn't work on instances that aren't constructed by Guice. We must call a method with AOP annotation on Guice proxy only (We cannot simply invoke this method from another method of this instance)



Guice AOP doesn't work particularly well on the Android platform while using Guice with Android applications, `guice-3.0-no_aop.jar`.

## Concerns related to performance

Guice generates proxies at runtime, which bring AOP mechanism in to the picture. It is an overhead, because it makes the application load slower. Also it adds overhead of increasing method invocations (each method invocation covered by an interceptor is preceded by a call to Guice proxy). These two concerns have been a subject matter of debate; how much overhead is acceptable against the convenience of simpler application design?

The real problem starts where application designers tend to solve problems with AOP, which could be solved without it. Such an increased AOP usage in design makes an application hard to debug and introduces performance-related problems.

Readers are advised to avoid unnecessary usage of AOP and try to figure out whether a concern is truly cross-cutting in nature before solving it with AOP.

## Summary

In this chapter, we have learned how to use AOP provided classes to solve cross-cutting concerns, such as Exception Handling, Logging messages, managing security, and transactions. We briefly discussed AOP conceptually, and also saw the various artifacts, which form the core of AOP. Later on, we discussed the Factory class Matcher, binding options from AbstractModule, and implementation of MatchingInterceptor. We also saw limitations regarding Guice and not using it on certain platforms.

# Prerequisites

To run the samples of various chapters accompanying this book, various software is required. Following are the various sections detailing their installation steps. Proper care is taken while providing instructions for various platforms like Mac OSX, Linux, and Windows. However, owing to various factors, such as different operating system versions, and dependency on third-party libraries, these may not work for every possible case. Reader is advised to visit the web pages (which are provided as a reference) in case any installation instruction doesn't work.

## Getting the sample code

Sample code for the book is available at [https://bitbucket.org/hussain-pithawala/begin\\_guice](https://bitbucket.org/hussain-pithawala/begin_guice). Readers are advised to clone the repository and work with the code samples. Also it is advised to make frequent changes in the code samples while reading various chapters.

## Installing Git

Git is a distributed version control system available free of cost. In case you are not familiar with Git, visit <http://git-scm.com/book/en/Getting-Started-Git-Basics> for a quick tutorial and <https://help.github.com/articles/set-up-git> for installing Git on your system.

## Cloning the repository

Once Git is installed, use the following command to clone the repository at a convenient location on your system.

```
shell> git clone https://bitbucket.org/hussain-pithawala/begin_guice.git
```

Invocation of following command would clone the repository locally at the `begin_guice/` directory. Readers can visit various subdirectories inside it, for chapter-specific sample code. For example, `begin_guice/chapter_1/flights` and `begin_guice/chapter_1/flights_with_guice` contain sample applications for *Chapter 1, Getting Started with Guice*.

## JDK

JDK 7 is preferred for running sample code, and is a prerequisite for installing and running **Maven**, **Eclipse**, and **Tomcat**. Readers are advised to visit <http://docs.oracle.com/javase/7/docs/webnotes/install> for detailed installation instructions about various platforms. In case JDK is already installed on your system, please proceed to install the rest of the prerequisites.

## Installing Maven

Download a binary release of Maven <http://maven.apache.org/download.html>; it is available in various formats, and most of the archive tools are able to extract them. Extract it in a convenient location, and follow rest of the procedure for different operating systems.

## Mac OSX and Linux


Assuming the expanded location of Maven archive to be `/usr/local/apache-maven-3.1.0`; execute the following commands in order to install Maven:

1. Create a soft link to the Maven installation. This would save you from changing the path variables in case you go for a version upgrade.

```
ln -s /usr/local/apache-maven-3.1.0 maven
```

2. Export the variable `M2_HOME`

```
export M2_HOME=/usr/local/maven
export PATH=${M2_HOME}/bin:${PATH}
```

[  Add these variables to your ~/.bash\_profile or ~/.bash\_login files to let you use Maven even when your current terminal session expires. ]

## Microsoft Windows


Assuming the expanded location of Maven archive to be c:\Program Files\apache-maven-3.1.0; execute the following commands in order to install Maven:

1. Set M2\_HOME variable

```
C:\> +set M2_HOME=c:\Program Files\apache-maven-3.1.0+
```

2. Add this variable to PATH

```
C:\> +set PATH=%PATH%;%M2_HOME%\bin+
```

[  Add these variables to system variables using Control Panel. This would let you run the Maven even when your current terminal session expires. ]

## Installing Eclipse

Download the latest version of eclipse from <http://eclipse.org/downloads>. Choose your operating system from the downloads page, and pick up a suitable download option, 32-bit or 64-bit, depending upon your operating system platform. Download it, and expand it at a convenient location, as a directory. This directory has an executable named eclipse, which could be directly executed for starting the eclipse IDE.

## Installing the m2eclipse plugin

Eclipse installations usually come with the m2eclipse plugin bundled with them. This assists users in developing the Maven projects easily, and even importing a pre-existing Maven project. In case you need to install it explicitly, Please visit <http://eclipse.org/m2e/> and use the update link <http://download.eclipse.org/technology/m2e/releases>. This is an eclipse IDE usable-link and should be used via the **Help** menu to install new software from eclipse.



## Installing Apache Tomcat

Though **Apache Tomcat** could be installed via package managers and installers available; we would follow a platform-independent approach, which gives the user more freedom to configure and administer it. Apache Tomcat requires the JDK installed on your system to work. Download the latest version .zip or .tar.gz version from <http://tomcat.apache.org/download-70.cgi>, and extract at a convenient location. Browse into the directory, and you will find various subdirectories. Here is a short description for these:

- `bin`: Location for the binary executables
- `conf`: Location for the configuration files
- `lib`: Location for the external libraries
- `logs`: Default location for the log files
- `temp`: Default location for the temporary files
- `webapps`: Location for web application or .war files deployment
- `work`: Contains generated application-specific servlet files for various JSPs

Tomcat is ready to use after this installation. The packaged .war files (present as code samples from Chapter 4 to Chapter 8) could be deployed here.

## Installing MySQL

MySQL is available freely (community edition) from <http://dev.mysql.com/downloads/mysql>. Choose the operating system and follow the instructions mentioned for it.

## Microsoft Windows

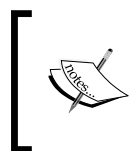
Pick up a convenient download option available for windows from the download section of MySQL website. Choose an option according to your platform (32-bit or 64-bit). Download the .zip file. Unzip the archive, and start the installation using `setup.exe` or `setup.msi` (Microsoft installer file). The installation wizard gives you various options like **Typical**, **Complete**, or **Custom**. Choose the one which suits your needs.

## Mac OSX

Pick up a convenient download option available for Mac OSX from the choices (for example `mysql-5.6.12-osx10.7-x86_64.dmg`). Click the `.dmg` file, and the setup installation process should start. Choose any of the available options available which suit your needs.

## Linux

Most of the Linux installations come with a pre-bundled MySQL. In case there is no pre-installed bundle present, download the distribution-specific file by choosing your Linux variant and proceed by launching the file.



A typical MySQL installation would provide you a server, a command-line client, and command-line utilities such as `mysqldump` and `myiasmchk`. These tools help in administering the MySQL server.

## Importing MySQL data

For running samples for Chapter 6, 7, and 8, the MySQL data dump has been provided in the file `flights.sql`. This file is present in the `begin_guice/` directory. In order to check out if the samples for these chapters are running, follow the given steps:

1. Start the MySQL server:  

```
shell> mysqld start
```
2. Login as a root user into MySQL server:  

```
shell> mysql -uroot -A
```
3. Create a database flights:  

```
mysql>create database flights;
```
4. Grant the permission to a user flight with password as flight:  

```
mysql>grant all on flights.* to 'flight'@'localhost' identified by 'flight';
```

5. Quit the MySQL console:  
`mysql> quit;`
6. Import the data into the flights table from shell:  
`shell> mysql flights -uflight -pflight < flights.sql`
7. Log back again with credentials for flight:  
`shell>mysql flights -uflight -pflight`
8. Check the data:  
`mysql>select * from flight;`

The execution of last command would show the data in the table `flight`.

Prior to running the samples from Chapter 6 to Chapter 8, make sure the MySQL server is running.

# Index

## Symbols

**@BindingAnnotation** annotation 24  
**@CheckedProvides** annotation 35  
**@ImplementedBy** annotation  
  about 21  
  interface, annotating 26  
**@Inject** annotated constructors 26  
**@Inject** annotation 12, 18  
**@Named** annotation 13, 25  
**@ProvidedBy** annotation 21  
**@Provides** annotation 32, 33  
**@RequestParameters** annotation 56  
**@RequestScoped** annotation 55  
**@Retention** annotation 24  
**@SessionScoped** annotation 55  
**@Target** annotation 24  
**@Transactional** annotation 71

## A

**accept()** method 82  
**acceptScopingVisitor()** method 88  
**ActionContextMap** 60  
**AnalyzeBindingVisitor** 88  
**annotation**  
  developing, for scope 44  
**AOP**  
  about 95  
  working 95, 96  
**AOP, with Guice**  
  about 96  
  limitations 101  
  LoggingInterceptor, implementing 96-98  
  matchers 99, 100

## Apache Tomcat

  URL, for installing 106

## ASM 96

**Aspect Oriented Programming.** *See* AOP

## Aspects 95

**AssistInstallModule** 80

**AssitedInject** 35-37

## B

### BCEL 96

**benefits, Elements SPI** 85

**binding annotations** 23, 24

**binding collections**

  about 37

  MultiBinder, using 38, 39

  TypeLiteral, using 38

**binding constants** 24

**binding language**

  about 51

  filters, mapping 53, 54

  servlets, mapping 52, 53

**binding properties** 25

**bindings**

  about 20

  configuring 19

**bindings, types**

  constructor bindings 21, 22

  instance bindings 21

  linked bindings 20

  untargeted bindings 21

## C

**characterstics, Struts 2** 61

**CheckedProviders** interface 33, 34

- classes, in guice-multibinder**
  - Multibinder 90
  - MultibindingsTargetVisitor 90
  - RealElement 90
  - RealMultibinder 90
- constructor binding** 21, 22
- constructor injection** 18
- control**
  - inverting, dependency injection used 9
- custom extension**
  - exploring 89
- custom scopes**
  - about 41
  - providers, modifying 42
  - writing 41

## D

- DefaultElementVisitor interface** 83
- dependencies**
  - about 7
  - initializing, directly 10, 11
  - injecting, in providers 31
  - inverting 8
  - resolving, directly 7
- Dependency Injection (DI)**
  - about 7, 9, 17
  - challenges 9
  - components 9
  - principles 12
  - used, for inverting control 9
- Dependency Inversion Principle**
  - URL 8
- DI container** 12
- direct binding** 43

## E

- eager singletons** 40
- early instantiation**
  - drawbacks 29, 30
- Eclipse**
  - about 104
  - URL, for installing 105
- EclipseLink** 68
- EDSL** 51

- Element interface** 83
- Elements SPI**
  - about 81, 83
  - benefits 85
  - implementing 84, 85
- ElementVisitor interface** 83
- EntityManager** 69
- EntityManagerFactory** 69
- Extensions SPI** 81, 86-88

## F

- features, Guice**
  - binding collections 29
  - providers 29
  - scoping 29
- field injection** 18
- filters**
  - mapping 53, 54
- FlightEngineModule** 75
- FlightJPASupplier** 75
- FlightSearchAction** 64, 65
- FlightSearch application**
  - client 73
  - FlightEngineModule 75
  - FlightJPASupplier 75
  - FlightServletContextListener 76
  - Persistence.xml 73
  - SearchRS 74
  - with Hibernate 3 73
  - with JPA 2 73
- FlightSearch application, with Struts 2**
  - about 63
  - FlightSearchAction 64, 65
  - FlightServletContextListener 63
  - index page 63
  - response page 65
- FlightServletContextListener** 63
- FlightServletContextListener** 76
- flights-struts-jpa-ext** 79
- flightsweb application**
  - structure 47-49
- FreeMarker** 62

## G

### Git

- about 103
- repository, cloning 104
- URL, for installing 103

### Guice

- about 7, 9
- AOP, supporting 96
- features 29
- integrating, with JPA 2 67

### Guice 3

- integrating, with Struts 2 59, 62

### Guice 3, and Struts 2 integration

- Struts2Factory 63
- Struts2GuicePluginModule 62

### Guice 3, with JPA 2

- @Transactional 71
- about 69
- JpaLocalTxnInterceptor 71
- JpaPersistModule 72
- PersistFilter 70
- PersistService interface 70
- UnitOfWork interface 70

### GuiceFilter

- using 49

### guice-multibinder 79, 89

### guice-persist 67

### Guice plugin

- about 79
- developing 79, 80

### Guice plugin dependency 80

### Guice SPI

- about 80, 81
- Elements SPI 81
- Extensions SPI 81

### guice-struts-2.3.0.jar file 59

## H

### HasDependencies interface 90

### Hibernate

- about 68
- URL 68

### Hibernate 3 68

## I

### index page 63

### injection

- types 18

### injections, types

- constructor injection 18
- field injection 18
- method injection 18

### injector 17

### injector binding 85

### instance binding 21

### interface

- annotating, with @ImplementedBy 26

### interfaces, in guice-multibinder

- HasDependencies 90
- Module 90
- MultibinderBinding 90
- ProviderWithExtensionVisitor 90

### Inversion of Control

- about 7, 8
- URL 8

## J

### JavaAssist 96

### Java Persistence API. *See* JPA

### JAXP 81

### JCE 81

### JDBC 81

### JDK

- about 104
- URL, for installing 104

### JDK 7 104

### JNDI 81

### JoinPoint 96

### JPA

- about 68
- URL 68

### JPA 2

- about 68
- Guice, integrating with 67

### JpaLocalTxnInterceptor 71

### JpaPersistModule 72

### JpaPersistService 70

**JSR 317 Expert Group** 68

**just in time bindings**

    @Inject annotated constructors 26

    about 26

    by default constructors 26

    interface, annotating with @ImplementedBy 26

## L

**linked binding** 20

**Linux**

    about 103

    Maven, installing 104

    MySQL, installing 107

**LoggingInterceptor**

    implementing 96-98

## M

**m2eclipse plugin**

    URL, for installing 105

**Mac OSX**

    about 103

    Maven, installing 104

    MySQL, installing 107

**matchers**

    about 95, 99

    for binding 99, 100

**Maven**

    about 14, 104

    installing, on Linux 104

    installing, on Mac OSX 104

    installing, on Microsoft Windows 105

    URL, for downloading binary release 104

**maven command** 59

**method injection** 18

**MethodInterceptors** 95, 97

**Microsoft Windows**

    Maven, installing 105

    MySQL, installing 106

**modifications, FlightEngineModule**

    about 43

    annotation, developing for scope 44

    binding, to annotation 44

    direct binding 43

**module bindings** 85

**Module interface** 90

**MultiBinder**

    using 38, 39

**MultibinderBinding interface** 90

**Multibinder class** 90

**Multibinder, in FlightEngineModule**

    working 91-93

**MultibindingsTargetVisitor class** 90

**MVC implementation**

    in Struts 2 61, 62

**MySQL**

    about 106

    installing, on Linux 107

    installing, on Mac OSX 107

    installing, on Microsoft Windows 106

**MySQL data**

    importing 107

## O

**object graph** 8

**Object Graph Navigation Language.** *See*  
    OGNL

**ObjectStack** 60

**Object to Relational mapping (ORM)** 68

**OGNL** 60

**OpenJPA** 68

## P

**PersistenceContext** 68

**persistence unit** 68

**Persistence.xml** 73

**PersistFilter** 70

**PersistService interface** 70

**pom.xml file** 47, 59

**processRequest() invocation** 29

**providers**

    @Provides annotation 32, 33

    about 29

    advantages 32

    dependencies, injecting in 31

    modifying 42

    need for 29, 30

    rolling 30, 31

    working 30

**ProviderWithExtensionVisitor interface** 90

## R

**RealElement class** 90  
**RealMultibinder class** 90  
**refactoring**  
    for using Guice 12, 13  
**response page** 65

## S

**sample applications**  
    building 14  
    code, compiling 15  
    running 15  
    unit tests, running 15  
**sample code**  
    URL, for obtaining 103  
**scope**  
    annotation, developing for 44  
    defining 43  
**scoping**  
    about 39  
    caution, exercising 57  
    custom scopes 41  
    eager singletons 40  
    singletons, with Guice 39, 40  
**SearchRequest object** 30  
**SearchResponse object** 30  
**SearchRS** 74  
**service** 81  
**service provider** 81  
**Service Provider Interface.** *See* SPI  
**ServletContextListener interface**  
    rolling out 49  
**ServletModule** 51  
**servlets**  
    mapping 52, 53  
**session-per-request** 69  
**session-per-transaction** 69  
**session strategies**  
    about 69  
    session-per-request 69  
    session-per-transaction 69

**singletons**  
    about 39  
    with Guice 39, 40  
**SPI** 80  
**Spring** 9  
**static injection** 27  
**Struts 2**  
    about 60  
    characterstics 61  
    Guice 3, integrating with 59, 62  
    MVC implementation 61, 62  
**struts2-core-2.3.14.jar file** 59  
**Struts2Factory class** 63  
**Struts2GuicePluginModule** 62

## T

**ThreadLocal object** 60  
**Tomcat** 104  
**TypeLiteral**  
    using 38

## U

**UnitOfWork** 70  
**unit tests**  
    running 15  
**unscoped provider** 43  
**untargeted binding** 21

## V

**value object**  
    about 30  
    SearchRequest 30  
    SearchResponse 30  
    URL, for info 30  
**ValueStack** 60  
**Velocity template** 62  
**visit() method** 82  
**Visitor Design pattern**  
    about 81, 82  
    URL 83



## W

**WAR (web application archive)** 47

**web scopes**

    @RequestParameters 56

    @RequestScoped 55

    @SessionScoped 55

    about 54

**web.xml file** 49

**Windows** 103

**wiring dependencies**

    about 7

    dependencies, inverting 8

    dependencies, resolving 7

    Inversion of Control 8



## Thank you for buying Learning Google Guice

### About Packt Publishing

Packt, pronounced 'packed', published its first book "*Mastering phpMyAdmin for Effective MySQL Management*" in April 2004 and subsequently continued to specialize in publishing highly focused books on specific technologies and solutions.

Our books and publications share the experiences of your fellow IT professionals in adapting and customizing today's systems, applications, and frameworks. Our solution based books give you the knowledge and power to customize the software and technologies you're using to get the job done. Packt books are more specific and less general than the IT books you have seen in the past. Our unique business model allows us to bring you more focused information, giving you more of what you need to know, and less of what you don't.

Packt is a modern, yet unique publishing company, which focuses on producing quality, cutting-edge books for communities of developers, administrators, and newbies alike. For more information, please visit our website: [www.packtpub.com](http://www.packtpub.com).

### About Packt Open Source

In 2010, Packt launched two new brands, Packt Open Source and Packt Enterprise, in order to continue its focus on specialization. This book is part of the Packt Open Source brand, home to books published on software built around Open Source licenses, and offering information to anybody from advanced developers to budding web designers. The Open Source brand also runs Packt's Open Source Royalty Scheme, by which Packt gives a royalty to each Open Source project about whose software a book is sold.

### Writing for Packt

We welcome all inquiries from people who are interested in authoring. Book proposals should be sent to [author@packtpub.com](mailto:author@packtpub.com). If your book idea is still at an early stage and you would like to discuss it first before writing a formal book proposal, contact us; one of our commissioning editors will get in touch with you.

We're not just looking for published authors; if you have strong technical skills but no writing experience, our experienced editors can help you develop a writing career, or simply get some additional reward for your expertise.



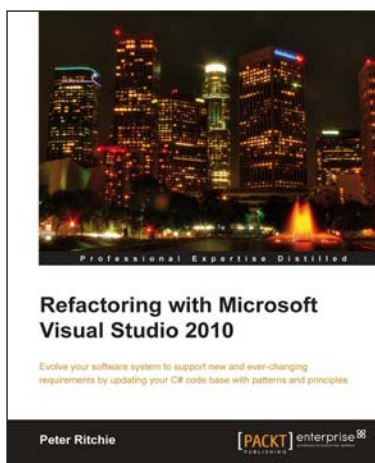
## Java EE 6 with GlassFish 3 Application Server

ISBN: 978-1-84951-036-3

Paperback: 488 pages

A practical guide to install and configure the GlassFish 3 Application Server and develop Java EE 6 applications to be deployed to this server

1. Install and configure the GlassFish 3 Application Server and develop Java EE 6 applications to be deployed to this server
2. Specialize in all major Java EE 6 APIs, including new additions to the specification such as CDI and JAX-RS
3. Use GlassFish v3 application server and gain enterprise reliability and performance with less complexity



## Refactoring with Microsoft Visual Studio 2010

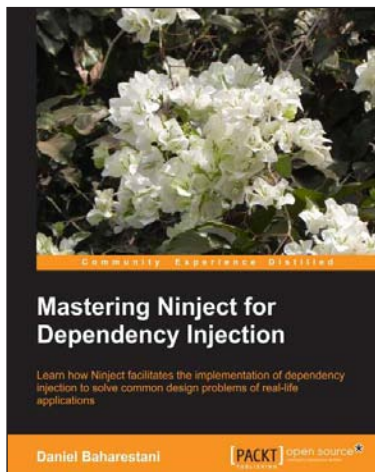
ISBN: 978-1-84968-010-3

Paperback: 372 pages

Evolve your software system to support new and ever-changing requirements by updating your C# code base with patterns and principles

1. Make your code base maintainable with refactoring
2. Support new features more easily by making your system adaptable
3. Enhance your system with an improved object-oriented design and increased encapsulation and componentization

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles



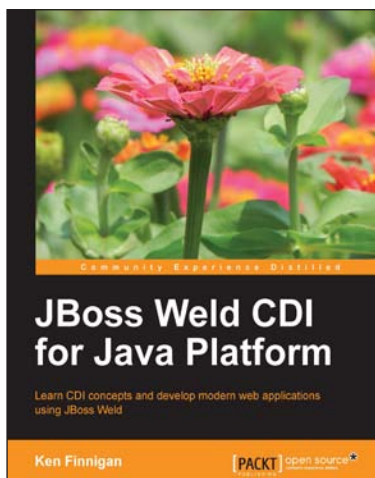
## Mastering Ninject for Dependency Injection

ISBN: 978-1-78216-620-7

Paperback: 126 pages

Learn how Ninject facilitates the implementation of dependency injection to solve common design problems of real-life application

1. Create loosely coupled applications by implementing dependency injection using Ninject
2. Learn how to design an enterprise application so as to maximize its maintainability, extensibility and testability
3. Automate the process of dealing with the dependencies of your application and object lifetimes



## JBoss Weld CDI for Java Platform

ISBN: 978-1-78216-018-2

Paperback: 122 pages

Learn CDI concepts and develop modern web applications using JBoss Weld

1. Learn about dependency injection with CDI
2. Install JBoss Weld in your favorite container
3. Develop your own extension to CDI
4. Decouple code with CDI events
5. Communicate between CDI beans and AngularJS.

Please check [www.PacktPub.com](http://www.PacktPub.com) for information on our titles