Chapter B

Errata

Hi guys, I am really thankful to all of you that have bought the book and have contributed to finding mistakes in it and in the code. Because of you, I have taken another careful look at the book questions and answers and decided this errata was needed.

B.1 Book corrections

Chapter 1: Introduction

The section **About the Certification Exam** needs an update on pages 3 and 4. Starting with January 2017 the exam is taken directly from home and the voucher is valid only three months. *(Submitted by Vittorio Marino)*

Chapter 2: Spring Bean LifeCycle and Configuration

In the section **Setter Injection** section, page 34 in the examples a bean named `simpleBean0` is declared, but not used as a dependency in the `complexBean`. The intended name, was obviously `simpleBean`. *(Reported by Gautham Kurup)*

! The original (erroneous) code sample is this:

```
<beans ...>
        <bean id="simpleBean0" class="com.ps.beans.SimpleBeanImpl"/>
        <bean id="complexBean" class="com.ps.beans.set.ComplexBeanImpl">
                <property name="simpleBean" ref="simpleBean"/>
        </bean>
</beans>
```

! And in future editions of the book, will be replaced with this:

```
<beans ...>
        <bean id="simpleBean" class="com.ps.beans.SimpleBeanImpl"/>
        <bean id="complexBean" class="com.ps.beans.set.ComplexBeanImpl">
                <property name="simpleBean" ref="simpleBean"/>
        </bean>
</beans>
```

The same bean is later referred in the paragraph explaining the behaviour.

! The original (erroneous) paragraph is this:

The code and configuration snippets above provide all the necessary information so that the Spring container can create a bean of type `SimpleBeanImpl`and a bean named `simpleBean0` and then inject it into a bean of type `ComplexBeanImpl` named `complexBean`.

! And in future editions of the book, will be replaced with this: The code and configuration snippets above provide all the necessary information so that the Spring container can create a bean of type `SimpleBeanImpl` and a bean named `simpleBean` and then inject it into a bean of type `ComplexBeanImpl` named `complexBean`.

Other small typos and mishaps.

Page	Original	Correction
19	`StubAbstractRepo<T extends AbstractEntity> implements +underlineAbstractRepo<T>`	`StubAbstractRepo<T extends AbstractEntity> implements AbstractRepo<T>`
29	Spring will help assembling the components a very pleasant job.	Spring makes assembling the components a very pleasant job.
43	Equivalent to `list.add(new SimpleBeanImpl).`	Equivalent to `list.add(new SimpleBeanImpl()).`
63	`for (String name : ctx.getAliases("sb02/sb03") ){ logger.info ("Alias for sb04 ->" + name); }`	`for (String name : ctx.getAliases("sb02/sb03") ){ logger.info ("Alias for` **sb02/sb03** `->" + name); }`
66	`INFO c.p.ApplicationContextTest - >> bye bye.`	Leftover log entry from code that was removed from the book.
70	The init method method must return void, have no arguments defined, and can have any access right...	The init method method should return void, because if a value is returned, it is ignored by the container, must have no arguments defined, and can have any access right...
72	The `InstantiatingBean` interface...	The **InitializingBean** interface...
74	... because the base-package| ...	... because the `base-package` ...
75	The methods that can be annotated with `@PostConstruct` must respect the same rules as every other init method: they must have no arguments, return void, and they can have any access right.	The methods that can be annotated with `@PostConstruct` must respect the same rules as every other init method: they must have no arguments, return any type (preferably void, because the returned value is ignored by the container anyway), and they can have any access right.
80	The destroy method must return void.	The destroy method can return any type, but as the container ignores the returned value, void is recommended.
98, 99, 101	Multiple typos: `@Autowire`	Should be corrected to: `@Autowired`
108	`@Cofiguration` (in "Stereotypes annotations" list)	`@Configuration` (in "Stereotypes annotations" list)

Table B.1: Corrections Table (part 1)

In **page 76** the developer is invited to do a practice exercise and add an initialisation method called `nitMethod2` and check to see how the log will change. The log printed afterwards assumes that the initialisation methods print the exact name of the "stage" they are executed in and thus the declaration of the `ComplexBean` in page 75 should be modified as follows(only the initialisation methods are depicted):

```
<beans ...">
    <bean id="simpleBean1" class="com.ps.sample.SimpleBean"/>
    <bean id="simpleBean2" class="com.ps.sample.SimpleBean"/>

    <bean id="complexBean" class="com.ps.sample.ComplexBean"
          c:_0-ref="simpleBean1"
```

```
            p:simpleBean2-ref="simpleBean2"
            init-method="initMethod2"
            destroy-method="destroyMethod"/>
    <context:component-scan base-package="com.ps.sample"/>
</beans>

//com.ps.sample.ComplexBean.java
public class ComplexBean {
...
    private void initMethod2() {
       logger.info(" --> Stage 4: Calling the initMethod2.");
    }

    @PostConstruct
    private void initMethod() {
        logger.info(" --> Stage 3: Calling the initMethod.");
        long ct = System.currentTimeMillis();
        if (ct % 2 == 0) {
            simpleBean2 = new SimpleBean();
        }
    }
    ...
}
```

(*Observation submitted by Süleyman Onur*)

In **page 90** a clarification regarding class `PropertySourcesPlaceholderConfigurer` is missing. This class is a specialization of `PlaceholderConfigurerSupport` (the same class extended by `PropertyPlaceholderConfigurer` that is used in previous examples) and thus is a bean definition post processor class. This class is designed as a general replacement for `PropertyPlaceholderConfigurer` starting with Spring 3.1. This class is used in section **2.1 Quick quiz**, page 113, question 9 as option B. (*Submitted by Edward Whiting*)

In **page 90** a configuration file is depicted that is confusing for some developers, because contains two configurations versions separated only by a comment.

```
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
        <!-- does not pickup @Configuration so the configuration
             class must be declared as a bean-->
        <context:annotation-config/>
        <bean class="com.ps.config.DataSourceConfig" />

        ~fvtextcolorred<!-- or the more practical way -->
        <!-- Picks up everything-->
        <context:component-scan base-package="com.ps.config"/>

</beans>
```

To make things clearer, in the following configuration snippet, two equivalent configurations file are depicted.

```
<!-- config-01.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>
```

```
        <!-- does not pickup @Configuration so the configuration
             class must be declared as a bean-->
        <context:annotation-config/>
        <bean class="com.ps.config.DataSourceConfig" />
</beans>

<!-- config-02.xml -->
<?xml version="1.0" encoding="UTF-8"?>
<beans ...>

        <!-- Picks up everything-->
        <context:component-scan base-package="com.ps.config"/>

</beans>
```

(*Observation submitted by Süleyman Onur*)

In section **Using Multiple Sources of Configuration**, **page 94**, in class `BootstrapTest` the context declaration is not quite correct. Because class `RequestRepoConfig` is annotated with `@Import(DataSourceConfig.class)`, specifying the `DataSourceConfig.class` as argument when creating the context is not really necessary and redundant. So the `setUp()` method should be modified to:

```
    ctx = new
         AnnotationConfigApplicationContext(RequestRepoConfig.class);
```

(*Observation submitted by Süleyman Onur*) Also, the `ctx` field declaration is missing.

```
public class BootstrapTest {
        private ApplicationContext ctx;
        ...
}
```

(*Submitted by Edward Whiting*)

In the section **Bean Naming section**, page 94, there is a huge error in the samples for `@AliasFor`. It was implied there that the value attribute of the `@Repository` can be aliased. At the time when the book was written, Spring 4.2 was under development and more information about the limits of the `@AliasFor` were published later on. Because autowiring by type was used, the tests still passed thus hiding the erroneous implementation. So there is a code sample and some text there which are erroneous, and should be skipped altogether.

! The original (erroneous) section is this:

And even more can be done. Aliases for meta-annotation attributes can be declared. Let's say that we do not like the value attribute name of the `@Repository` annotation and we want in our application to make it more obvious that the `@Repository` annotation can be used to set the id of a bean, by adding an id attribute that will be an alias for the value attribute. The only conditions for this to work are that the annotation declaring the alias has to be annotated with the meta-annotation, and the `annotation` attribute must must reference the meta-annotation.

```
package com.ps.sample;
import org.springframework.core.annotation.AliasFor;

@Repository
public @interface MyRepoCfg {
        @AliasFor(annotation = Repository.class, attribute = "value")
        String id() default "";
        }
}
```

Thus, the repository beans can be declared as depicted in the following code snippet:

```
...
import com.ps.MyRepoCfg;
...

//JdbcRequestRepo. java bean definition
@MyRepoCfg(id = "requestRepo")
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
implements RequestRepo{
...
}
```

! And in future editions of the book, will be replaced with this:

And even more can be done. Aliases for meta-annotation attributes can be declared. Let's say that we do not like the `lTout` attribute name of the `@DsCfg` annotation and we just want to refer to it differently. The only conditions for this to work are that the annotation declaring the alias has to be annotated with the meta-annotation, and the annotation attribute must must reference the meta-annotation.

```
package com.ps;
import org.springframework.core.annotation.AliasFor;
@DsCfg
public @interface MyDsCfg {
        @AliasFor(annotation = DsCfg.class, attribute = "lTout")
        int timeout() default 200;
}
```

Thus, we can now apply the new annotation on the repository beans, that can be declared as depicted in the following code snippet:

```
...
import com.ps.MyDsCfg;
...
//JdbcRequestRepo. java bean definition
@MyDsCfg(timeout = 2400)
@Repository
public class JdbcRequestRepo extends JdbcAbstractRepo<Request>
        implements RequestRepo{
...
}
```

! `@AliasFor` cannot be used on any stereotype annotations. The reason is that the special handling of these value attributes was in place years before `@AliasFor` was invented. Consequently, due to backward compatibility issues it is simply not possible to use `@AliasFor` with such value attributes. When writing code to do just so (aliasing value attributes in stereotype annotations), no compile errors will be shown to you, and the code might even run, but any argument provided for the alias will be ignored. the same goes for the `@Qualifier` annotation as well.

This information about the `@AliasFor` annotation can be found here: https://github.com/spring-projects/springframework/wiki/Spring-Annotation-Programming-Model#1-can-aliasfor-be-used-with-the-value-attributes-for-componentand- qualifier. [1]
The sources has also been updated on the raw repository: `https://github.com/iuliana/pet-sitter`

[1] The URL does not fit the page, but for electronic use the clickable URL is this: FAQ @AliasFor

Other small typos and mishaps.

Page	Original	Correction
103	The `@Autowired` annotation by default requires the dependency to be mandatory, but this behavior can be changed, by setting the required attribute to `true:`	The `@Autowired` annotation by default requires the dependency to be mandatory, but this behavior can be changed, by setting the required attribute to **false:**
105	Table 2-3. *Prefixes and corresponding paths*	Table 2-3. **Spring and JSR 250 annotation equivalents**
120	SimpleAbstractService<Pet>	AbstractRepo<T>

Table B.2: Corrections Table (part 2)

Also in section **2.1 Quick quiz**, question 8, the annotation `@Autowired` is missing from the setter in option **A.** ! The corrected option looks like this:

```
@Component
public class QuizBean {

@Autowired
 public void setPetitBean(PetitBean petitBean) {
        this.petitBean = petitBean;
    }

  @PostConstruct
  private void initMethod() {
        logger.info("--> I'm calling it bean soon");
  }
}
```

(*Submitted by Peter Jurkovic*)

In section **2.1 Quick quiz**, page 114, question 11 is missing a piece of code that would make one of the answers a valid one. The respective piece of code is depicted below:

```
package com.cfg;

import com.com.another.quiz.AnotherQuizBean;
import org.springframework.context.ApplicationContext;
import org.springframework.context.annotation.
    AnnotationConfigApplicationContext;

public class AppConfigTest {

    public static void main(String... args) {
        ApplicationContext ctx =
             new AnnotationConfigApplicationContext(AppConfig.class);
        AnotherQuizBean quizBean =
            ctx.getBean("anotherQuizBean", AnotherQuizBean.class);
        if(quizBean != null) {
            System.out.println("All is well!");
        }
    }
}
```

(*Observation submitted by Süleyman Onur*)

Chapter 3: Testing Spring Applications

Small typos and mishaps. (*Observations submitted by Süleyman Onur*)

Page	Original	Correction
133	`InjectMock` in the first paragraph	should be replaced with `@InjectMocks`
144	When classes are annotated with this anclasses become...	When classes are annotated with this **annotation, classes** become
145	`public DataSource dataSource() throws SQLException {`	`public DataSource` **dataSource()** `throws SQLException {`
147	option B: `setUp`	should be replaced by `initMocks`
147	option A: `MockitoRunner`	should be replaced by `MockitoJUnitRunner` and method `initMocks` should be removed because it is redundant

Table B.3: Corrections Table (part 3)

Chapter 4: Aspect Oriented Programming with Spring

Other small typos and mishaps. In page 178, in the **Summary** section there is an affirmation that is incomplete:

Page	Original	Correction
160	`return new HashSet<>(jdbcTemplate.query(sql, new Object{username}, rowMapper));`	`return new HashSet<>(jdbcTemplate.query(sql, new Object[]{username}, rowMapper));`

160	`import org.aspectj.lang.annotation.Component;`
	`import org.aspectj.lang.annotation.Aspect;`
	`import underlineorg.aspectj.lang.annotation.Before;`
		import org.springframework.stereotype.Component;
		`import org.aspectj.lang.annotation.Aspect;`
		import org.aspectj.lang.annotation.Before;

160	`@Before("execution(public com.ps.repos .'JdbcTemplateUserRepo+ .findById(..))")`	`@Before("execution(public * com.ps.repos.* .JdbcTemplateUserRepo+ .findById(..)))"`
165	`pointcut="execution(public com.ps.repos .'JdbcTemplateUserRepo+ .findById(..))"`	`pointcut="execution(public * com.ps.repos.* .JdbcTemplateUserRepo+ .findById(..))"`
166	`execution(public (public * com.ps.service.*.*Service+.*(..)`	`execution(public * com.ps.service.*.*Service+.*(..)`
166	`public * com.ps.repos .*.JdbcTemplateUserRepo+ .findById(..)) && +underlinewithin(com.ps.*)`	`public * com.ps.repos .*.JdbcTemplateUserRepo+ .findById(..)) && within(com.ps.*)`
171 172	`if (StringUtils.indexOfAny(pass, new String{"$", "#", "$", "%"}) != -1)`	`if (StringUtils.indexOfAny(pass, new String[]{"$", "#", "%"}) != -1)`
182	`execution(* update(Logn,..))`	**execution(* update(Long,..))**
182	B. an expression to identify methods to which the advice applies to	B. an expression to identify methods to which the advice applies
183	**Question 8**: What is a pointcut?	**Question 8**: What is a pointcut? **(choose all that apply)**

Table B.4: Corrections Table (part 3)

Only public Join Points can be advised (you probably suspected that). This is not true when CGLIB is on the classpath. In this case the proxy type will be a subclass of the target object class, and this means protected methods can be intercepted as well. *(Observation submitted by Farid Guliyev)*

Chapter 5: Data Access

Other small typos and mishaps.

Page	Original	Correction
193	`public JdbcTemplate ~underlinejdbcTemplate()`	`public JdbcTemplate` **jdbcTemplate()**
205	`new Object{username}, rowMapper));`	`new Object[]{username}, rowMapper));`
211	Figure 5.8 Bad caption: Created by a `EntityManagerFactoryBean`	Figure 5.8 Correct caption: Created by a `FactoryBean<EntityManagerFactory>`
215	Since the transactional behaviour must be propagated to the repository, the repository class or methods are annotated with `@Transactional` too. (Submitted by *Marek Muratow*)	The transactional behaviour is automatically propagates from the service to the repository so there is no need to annotate them both, unless you want each of the methods to be executed in different transactions.
224	`@SqlConfig(transactionMode = SqlConfig. TransactionMode.ISOLATED)`	`@SqlConfig(transactionMode = SqlConfig.TransactionMode.ISOLATED, transactionManager="txMng")`
225	READ_COMMITTED: ... repeatable read ...	READ_COMMITTED: ... **non-repeatable** read
225	The `@Transactiona` annotation...	The `@Transaction` annotation...
226	The `@Before` annotated method...	The `@BeforeTransaction` annotated method...
227	`<aop:pointcut ... >`	`<aop:pointcut ... />`
227	`<aop:advisor pointcut-ref= "userService"`	`<aop:advisor pointcut-ref= "allMethods"`
227	`<tx:advice id= "transactionalAdvice"> <tx:atributes>`	`<tx:advice id= "transactionalAdvice">` **<tx:attributes>**
227	Task TODO 31 ... to **mange** transactions.	Task TODO 31 ... to **manage** transactions.
228	`@Tranactional`	`@Transactional`
242	`deleteById(user)`	`delete(user)`
241	`List<Users> list = session .createQuery("from User u where username= ?")`	`List<Users> list = session()` `.createQuery("from User u where username= ?")` **.setParameter(0, username).list();**
249	`public class TestDataConfig {@Bean...`	`public class TestDataConfig { ...`
263	Every `RepoMongoRepositorysitory` interface...	Every **MongoRepository** interface...
268	Q17, Option B bad written `@ RepositoryDefinition` annotation	Q7, Option B: `@RepositoryDefinition`

Table B.5: Corrections Table (part 4)

Starting with page 234, distributed transactions are mentioned sometimes as *global transactions*, they are one in the same, in case this was not made obvious in the book. They are also named XA and that is why *Using distributed transactions requires a JTA and specific XA drivers*. Also footnote 13 contains a link that does not work. The correct link is `https://www.javaworld.com/article/2077963/open-source-tools/distributed-transactions-in-spring--with-and-without-xa.html`

In page 220, in the nested transaction example, there is a message in the log that needs more explaining

`# Not actually nested.` That message is necessary because the current version of H2 when this book is being written does not support nested transactions.

In page 222 there is a code snippet used as example for the `rollbackFor` attribute that is not correct.

! The original (erroneous) snippet is this:

```
@Transactional(rollbackFor = MailSendingException.class)
@Override
public int updatePassword(Long userId, String newPass)
          throws MailSendingException {
    User u = userRepo.findById(userId);
    String email = u.getEmail();
    sendEmail(email);
    return userRepo.updatePassword(userId, newPass);
}
```

! The problem with this code snippet is that the exception is thrown before calling `userRepo.updatePassword(..)` to execute the actual operation that we are interested in. And in future editions of the book, will be replaced with this:

```
@Transactional(rollbackFor = MailSendingException.class)
@Override
public int updatePassword(Long userId, String newPass)
           throws MailSendingException {
    User u = userRepo.findById(userId);
    String email = u.getEmail();
    int result = userRepo.updatePassword(userId, newPass);
    sendEmail(email);
    return result;
}
```

Same goes for the example in section **Spring Programatic Transaction Model**, page 233, where the code below:

```
try {
        User user = userRepo.findById(userId);
        String email = user.getEmail();
        sendEmail(email);
        return userRepo.updatePassword(userId, newPass);
} catch (MailSendingException e) {
        status.setRollbackOnly();
}
```

Should be replaced with:

```
try {
        int result = userRepo.updatePassword(userId, newPass);
        User user = userRepo.findById(userId);
        String email = user.getEmail();
        sendEmail(email);
        return result;
} catch (MailSendingException e) {
        status.setRollbackOnly();
}
```

In pages 207-208 ideas regarding the Spring Data Exceptions are a little redundant. All paragraphs before *As you can notice from Figure 5-5 ..* will be replaced in future versions of the book with:

When working with data sources, there might be particular cases when things do not go well: sometimes, operations might be called on records that no longer exist; sometimes records cannot be created because of some database restrictions; sometimes the database takes too long to respond; and so on. In pure JDBC, the same type of checked exception is always thrown, `java.sql.SQLException`, and in its message there is an SQL code that can be used by the developer to identify the real cause and handle the situation accordingly. Being a checked exception it forces developers to catch it and handle it or declare it as being thrown over the call method call hierarchy. This introduces a form of tight coupling.
In Spring, the data access exceptions are unchecked - they extend `java.lang.RuntimeException`, which is why this class is exhibited in the previous image with a red border and can be thrown up the method call hierarchy to the point where they are most appropriate to handle. This is obviously a more practical way for the developer to handle database access exceptions without knowing the details of the data access API. The Spring data access exceptions hide the technology used to communicate with the database and contain human-readable messages that point exactly to the cause of the problem.
A simplifiedversion of the Spring data access exception hierarchy can be seen in Figure 5-5.
`<Figure 5-5 here>`
The Spring Data Access exceptions are thrown when `JdbcTemplate`, `JPA`, `Hibernate`, etc. are used. That is probably why they are part of the `spring-tx` module. Under the hood, there are translator components that can be used to transform any type of data access exceptions into Spring-specific exceptions, which is quite useful when one is migrating from one database type to another. For example, the `JdbcTemplate` instance takes care of transforming cryptic `java.sql.SQLExceptions` into clear and concise `org.springframework.dao.DataAccessException` implementations[2] using an infrastructure bean of a type that implements `org.springframework.jdbc.support.SQLExceptionTranslator`.

In page 267 question 10, the quiz solution reads A,B,D,E,F. But one of the interfaces in the answer is never mentioned in the book: `TransactionDefinition`. *(Observation by Patrick Dobner)*Thus we are compensating for that here. The **Spring Programatic Transaction Model** section is missing the following paragraphs and code samples.

Another way of using transactions programmatically is to use the transaction manager directly. Spring's abstract transaction management unit is the `PlatformTransactionManager` interface. Regardless of the transaction manager framework provider used in an application a bean of type implementing `PlatformTransactionManager` can be injected and used directly. This interface defines three methods depicted in the code snippet below:

```
package org.springframework.transaction;

public interface PlatformTransactionManager {
   TransactionStatus getTransaction(TransactionDefinition definition)
         throws TransactionException;

   void commit(TransactionStatus status) throws TransactionException;

   void rollback(TransactionStatus status) throws TransactionException;
}
```

The `getTransaction(..)` returns a currently active transaction or creates and returns a new one, all depending on the transaction manager configuration. The parameter of type `TransactionDefinition` encapsulates transaction

[2]Class *org.springframework.dao.DataAccessException* is an abstract class, parent of all the family of Spring Data Access exceptions.

properties like isolation level, propagation behaviour, timeout, etc. It returns an object representing a transaction status of type implementing Spring's `TransactionStatus` interface. This object is used by the transaction manager to modify the transaction object.
The `commit(..)` and `rollback(...)` methods purpose is obvious because of their naming, they are the methods used by the transaction manager to change transaction status, depending on the success or failure of a transaction.
The code snippet below depicts the previous code sample, written by using the transaction manager bean directly. As you can see the `PlatformTransactionManager` is autowired by Spring and can be used to commit or rollback a transaction.

```
package com.ps.services.impl;

...
import org.springframework.jdbc.core.JdbcTemplate;
import org.springframework.transaction.PlatformTransactionManager;
import org.springframework.transaction.TransactionDefinition;
import org.springframework.transaction.TransactionStatus;
import org.springframework.transaction.support.DefaultTransactionDefinition;
import javax.sql.DataSource;

@Service("programaticUserService2")
public class ProgramaticUserService2 implements UserService {

    private JdbcTemplate jdbcTemplate;
    private PlatformTransactionManager transactionManager;

    @Autowired
    public ProgramaticUserService2(DataSource dataSource,
          PlatformTransactionManager txManager) {
        this.transactionManager = txManager;
        this.jdbcTemplate = new JdbcTemplate(dataSource);
    }

   @Override
    public int updatePassword(Long userId, String newPass)
         throws MailSendingException {
        TransactionDefinition def = new DefaultTransactionDefinition();
        TransactionStatus status = transactionManager.getTransaction(def);

        String sql = "update P_USER set PASSWORD=? where ID = ?";
        int result;
        try {
            result  = jdbcTemplate.update(sql, new Object[]{newPass, userId});

            sql = "select u.ID as ID, u.USERNAME as USERNAME," +
                       " u.USER_TYPE as USER_TYPE," +
                       " u.EMAIL as EMAIL, u.PASSWORD as PASSWORD" +
                       " from P_USER u where u.ID = ?";
            User user = jdbcTemplate.queryForObject(sql,
                       new Object[]{userId}, new UserRowMapper());
            transactionManager.commit(status);
            String email = user.getEmail();
            sendEmail(email);
```

```
            return result;
        } catch (MailSendingException e) {
            status.setRollbackOnly();
        } catch (Exception e) {
            // do rollback for any exception except MailSendingException
            transactionManager.rollback(status);
            throw e;
        }
        return 0;
    }
   ...
}
```

Chapter 6: Spring Web

In page **290**, in the depiction of the `WebInitializer` class, all **[]** are missing, all because of a formatting issue. The correct version of the class, is contained in the sample code attached to the book, nevertheless I will add here the correct version of the class.

```
package com.ps.config;

import org.springframework.web.servlet.support.
     AbstractAnnotationConfigDispatcherServletInitializer;
...
public class WebInitializer extends
     AbstractAnnotationConfigDispatcherServletInitializer {
    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class<?>[]{
                ServiceConfig.class
        };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class<?>[]{
                WebConfig.class
        };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[]{"/"};
    }

    @Override
    protected Filter[] getServletFilters() {
        CharacterEncodingFilter cef = new CharacterEncodingFilter();
        cef.setEncoding("UTF-8");
        cef.setForceEncoding(true);
        return new Filter[]{new HiddenHttpMethodFilter(), cef};
    }
}
```

Other small typos and mishaps.

Page	Original	Correction
300	Image 6-14 explanation list, item 2: `Authenticaltion Manager`	Image 6-14 explanation list, item 2: `Authentication Manager`
304	**Table 6-2.** Spring Security chained filters	**Table 6-2.** Spring Security chained filters and their positions
304	`@Override protected void configure (HttpSecurity http) throws Exception { http.addFilterAfter( SecurityContextPersistenceFilter.class, customConcurrencyFilter());`	`@Override protected void configure (HttpSecurity http) throws Exception { http.addFilterAfter( customConcurrencyFilter(), SecurityContextPersistenceFilter.class);`
308	using CSFR tokens ...	using CSRF tokens...
311	`<!-- spring-config.xml -->`	<!– security-config.xml –>
316	**Table 6-3.** Spring Security chained filters	**Table 6-2. CSRFTokenRepository Spring Implementations**
318	`Override protected void configure`	`@Override protected void configure`
326	In table 6-4, *jane does*	should be **jane doe**
331 335 342	**[]** are missing, all because of a formatting issue	should be: `String[] beanNames = ctx.getBeanDefinitionNames();`
338	`Server: port: 9000`	`server: port: 9000`
339	`app: port: 8084`	`app: port: 9000`
345	Question 1: ...	Question 1: ... **(choose all that apply)**
347	Question 11,12: ...	Question 11,12: ... **(choose all that apply)**

Table B.6: Corrections Table (part 5)

Chapter 7: Spring Advanced Topics

In page **401**, **Table 7-4** has a wrong caption and a wrong entry for the GET method. In page **405** the `WebConfig` class is missing the `@` from the `@Configuration` annotation.

HTTP Method	@RequestMapping	Spring 4.3 Annotation
GET	@RequestMapping(method=RequestMethod.**GET**)	@GetMapping
POST	@RequestMapping(method=RequestMethod.POST)	@PostMapping
DELETE	@RequestMapping(method=RequestMethod.DELETE)	@DeleteMapping
PUT	@RequestMapping(method=RequestMethod.PUT)	@PutMapping

Table B.7: HTTP Methods Spring annotations

In page **433** question 5, option C: *annotates* should be **annotated**.

Chapter 8: Spring Microservices with Spring Cloud

Page	Original	Correction
437	figre	**figure**

446	`# Discovery Server Access`
	`eureka:`
	`  client:`
	`  registerWithEureka: false`
	`  fetchRegistry: true`
	`  ...`
		`# Discovery Server Access for pets-service`
		`eureka:`
		`  client:`
		`  registerWithEureka: true`
		`  fetchRegistry: false`

450	`${ipAddress}:${spring.application.name` `  :${server.port}}`	`${ipAddress}:${spring.application.name}` `  :${server.port}`
456	This is a Spring Boot annotation that "under the hood" configures a LocalContainerEntityManagerFactory-Bean and sets the packagestoScan property ...	This is a Spring Boot annotation that "under the hood" configures a LocalContainerEntityManagerFactory-Bean and sets the **packagesToScan** property ...

Table B.8: Corrections Table (part 6)

Appendix A

In the Appendix A, the quiz solution section for Chapter 2, at editing, the numbering of the answers was misplaced, so the number of the questions and the answers, do not fit. There are also small errors in the answers to questions 4 and 8. The full list of answers with proper numbering and corrections is depicted below:

Quiz Solution for Chapter 2

1. **Answer:** A, B, C
2. **Answer:** B
3. **Answer:** A, B,D (C, Interface-based injection is not supported in Spring. D, field-based injection is supported by annotating fields with `@Autowired`, `@Value` or related annotations; JSR-250 `@Resource`, JSR-330 `@Inject`.[3])
4. **Answer:**
   Original: *Answer: A, B, C (As stated in Chapter 2)*
   Correct: **A** (`ClassPathXmlApplicationContext` is a simple convenience `ApplicationContext` implementation that can be used to load the definitions from the given XML files and automatically refreshing the context.)
5. **Answer:**
   Original: B (Only `@Component` is contained in the `org.springframework.stereotype package`)
   Correct: **A** (**Only** `@Component` **is contained in the** `org.springframework.stereotype package`)

[3] Spring Framework Reference: `http://docs.spring.io/spring/docs/4.2.3.RELEASE/spring-framework-reference/htmlsingle/#beans-annotation-config`

(The comment pointed to the correct answer, the letter was incorrect. Many thanks to Tobias Hochgürtel for noticing this.)

Clarification: Many readers have submitted questions about this answer asking why `@Configuration` is not considered a stereotype annotation. According to the Spring Reference Documentation we could consider that `@Configuration` is a marker for any class that fulfills the role or stereotype of a configuration class. But it is not explicitly mentioned as such in the documentation `https://docs.spring.io/spring/docs/4.3.x/spring-framework-reference/htmlsingle/#beans-stereotype-annotations`. Any new annotation using `@Component` is not necessary automatically a new stereotype. Sure, `@ComponentScan`is able to detect it according a technical perspective because it has `@Component`, but semantically would has no sense, because a bean of type configuration will rarely be used for anything else than bean declaration.
Thus, `@Configuration` is in the edge in some way. It can be considered a kind of stereotype, but only for infrastructure.

6. **Answer:** A

7. **Answer:** C

8. **Answer:**
   Original: *Answer: A,B,C*
   Correct: **A**.
   C is not a good answer, because of the `p:petitBean` definition. The `quizBean` has a dependency on the `petitBean` bean, thus a reference to it is needed. The equivalent bean definition, that would have made the original answer correct is:
   B is not a good answer either, because the `petitBean` property is injected using a constructor not a setter. So the bean definitions are not equivalent, even if the resulting beans are. (*Observation submitted by Süleyman Onur*)

```
<bean id="quizBean" class="QuizBean"
                init-method="initMethod" p:petitBean-ref="petitBean" />
```

9. **Answer:** A bean factory post processor modifies bean definitions before creating beans. This is why **B.** `PropertySourcesPlaceholderConfigurer` is correct. **C.** is not correct because `CommonAnnotationBeanPostProcessor` is the type of bean that enables modifications of beans, so after a bean is created from a bean definition, this type of bean allows for annotations like `@PostConstruct` and `@PreDestroy` to be picked up.

10. **Answer:** C

11. **Answer:** B

12. **Answer:** A

Quiz Solution for Chapter 3

Original: Answer: A,D
Correct: D. `@RunWith(MockitoRunner.class)` is not required, because mocks are initialized in the constructor.

Quiz Solution for Chapter 4

The answer for **Question 1** contains a mistake. (*Submitted by Edward Whiting*)
Original: Answer: B, C, D
Correct: A,B,C,D (connecting to a database is also a cross-cutting concern)

The answer for **Question 2** is wrong. (*Submitted by Edward Whiting*)
Original: Answer: A, B, C, D, E
Correct: B, D, E

! The three annotations `@Before`, `@After` and `@AfterReturning` are used to define when the advice applies. From a syntactical point of view, an advice is actually a method. What qualifies a method as an advice is the presence of one of those annotations. And this is what *declare* actually means in the context of this question.
Just keep in mind, when actually writing code, if `@Aspect` is missing on the class where the advice is defined and there is no `@Pointcut` to match where should the advice be applied, at runtime the advice methods are ignored.

The answer for **Question 4** is ambiguous and needs clarification. (*Submitted by Nicola Viola*)
The answer for the question is indeed *A. none: Spring AOP supports only advising public methods*, **but only when CGLIB is not used.** In the official documentation we find the following: *With CGLIB, public and protected method calls on the proxy will be intercepted, and even package-visible methods if necessary. However, common interactions through proxies should always be designed through public signatures.*[4] Without CGLIB, the proxy is an object that wraps around the target object and the proxy type is a class (generated by Spring IoC at runtime) that implements the same interface as the target object. Taken this into consideration, to make sure the proxy object can call methods of the target object, (the methods can be advised) the methods in the target object should be public as well. Because if the methods are protected, the Spring generated proxy won't be able to call them. This is obviously not the case when CGLIB is used, as the proxy class extends the class of the target object, and protected methods can be accessed and thus advised.

The answer for also **Question 8** contains a mistake. *(Reported by Patrick Dobner)*
Original: Answer: A, B
Correct: **B** (an expression to identify methods to which the advice applies to), **C** (a predicate used to identify join points)

Sample Exam

Question 1 has a (choose one) after the question content, which is confusing as the answer is made of more than one option. *(Reported by Tibor Kalman)*
Original: Which of the following affirmations is false? (choose one)
Correct: Which of the following affirmations is false?

Question 3 has an incorrect numbering of the answers.
Original:
Are the following two bean declaration equivalent?

A.
```
(1)
<bean id="someBean" p:someValue-ref="someOtherBean" />

(2)
@Component("someBean")
public class SomeBean {
        private SomeBean someValue;

        @Autowired
        @Qualifier("someOtherBean")
```

[4] Reference: `https://docs.spring.io/spring/docs/4.3.9.RELEASE/spring-framework-reference/htmlsingle/#aop-pointcuts-designators`

```
            public setSomeValue(SomeBean arg){
                    this.someValue = arg;
            }
    }
```

yes

B. no

C. the second is not a valid bean definition

Correct: Are the following two bean declarations equivalent?

```
(1)
<bean id="someBean" p:someValue-ref="someOtherBean" />

(2)
@Component("someBean")
public class SomeBean {
        private SomeBean someValue;

        @Autowired
        @Qualifier("someOtherBean")
        public setSomeValue(SomeBean arg){
                this.someValue = arg;
        }
}
```

A. yes

B. no

C. the second is not a valid bean definition

Sample exam, Answers Question 3 is missing the `void` return type for the `setSomeValue(...)` set method.

Question 23
Original: *C*
Correct: *B,C*

In the Answers section, the answer for question 23 is wrong.
Original: option **B**: implement `disposableBean`
Correct: option **B**: implement `DisposableBean`

In the Answers section, the answer for question 30 is wrong.
Original: *30. A*
Correct: **30. B**

B.2 Code updates and observations

In the 1.4.1 version of the Gretty plugin the `port` property was renamed to `httpPort`. Thus, old Gradle configurations like the one depicted below will cause a build failure.

```
gretty {
    port = 8080
    contextPath = '/mvc-layout'
}
```

The configuration must be corrected and the `port` property must be replaced with `httpPort`

```
gretty {
    httpPort = 8080
    contextPath = '/mvc-layout'
}
```

In case you want more information, or keep track of the issue I created related to this, take a look on the Gretty plugin GitHub page: `https://github.com/akhikhl/gretty`

Most small typos and formatting corrections were submitted by *Marco Pelucchi* and *Edward Whiting*. Thank you both for using your attention to detail to make this book better!