

# Seasar2 Reference Documentation

## DI Container with AOP

Version: 2.4

---

# 目次

I. Seasar2の紹介	1
1. 概要	2
1.1.	2
1.2.	2
2. セットアップ	3
2.1.	3
3. クイックスタート	4
3.1. はじめに	4
3.2. はじめてのSeasar2アプリケーション	4
3.3. プロパティの設定	5
3.4. DIの導入	7
3.5. AOPの適用	8
3.6. まとめ	10
4. アーキテクチャ	11
4.1.	11
II. S2コンテナ	12
5. 概要	13
5.1.	13
5.2.	13
6. Dependency Injection	14
6.1. はじめに	14
6.2. Dependency Injection概要	14
6.3. Dependency Injectionのタイプ	14
6.4. ライフサイクルとDI	14
6.5. 自動バインディング	15
6.6. 特殊なコンポーネント	15
6.7. 外部バインディング	15
6.8. まとめ	15
7. Aspect-Oriented Programming	16
7.1.	16
7.2. インターセプタの作成	16
7.3. インタータイプの作成	16
7.4. 標準インターセプタ	16
7.5. 標準インタータイプ	16
8. S2コンテナの作成	17
8.1.	17
8.2.	17
9. S2コンテナの利用	18
9.1.	18
9.2.	18
10. シングルトンS2コンテナ	19
10.1.	19
10.2.	19
11. 外部コンテキスト	20
11.1.	20
11.2.	20
III. 設定	21
12. 概要	22
12.1.	22
12.2.	22
13. diconファイル	23

13.1. はじめに	23
13.2. 基本構造	23
13.3. 文書型宣言	23
13.4. <components>要素	25
13.5. <description>要素	26
13.6. <meta>要素	26
13.7. まとめ	27
14. コンポーネント定義	28
14.1. はじめに	28
14.2. コンポーネント	28
14.3. <component>要素	28
14.4. まとめ	32
15. DI定義	33
15.1. はじめに	33
15.2. <arg>要素	33
15.3. <property>要素	33
15.4. <initMethod>要素	34
15.5. <destroyMethod>要素	35
15.6. まとめ	36
16. AOP定義	37
16.1. はじめに	37
16.2. <aspect>要素	37
16.3. <interType>要素	37
16.4. まとめ	38
17. OGNL	39
17.1.	39
17.2.	39
18. アノテーション	40
18.1.	40
18.2.	40
19. diconファイルの分割と編成	41
19.1.	41
19.2.	41
20. SMART deploy	42
20.1.	42
20.2.	42
21. コンポーネントの自動登録	43
21.1.	43
21.2.	43
22. S2コンテナのカスタマイズ	44
22.1.	44
22.2.	44
IV. Web	45
V. データアクセス	46
VI.	47
VII. テスト	48
VIII. for Framework Programmer	49
索引	50

---

# パート Ⅰ. Seasar2の紹介

---

# 第1章 概要

1.1.

1.2.

---

## 第2章 セットアップ

### 2.1.

# 第3章 クイックスタート

## 3.1. はじめに

S2Containerは、Dependency Injection（以降DIと略します）とAOP（Aspect-Oriented Programming）をサポートする軽量コンテナです。DIとは、複数のコンポーネント間の直接的な依存関係を排除し、それぞれのコンポーネントはインタフェースを通じて対話（コラボレーション）しようという考え方です。AOPとは、コンポーネントに特有の本質的な機能（core concern）と、ロギングなど複数のコンポーネントにまたがる機能（cross-cutting concern）とを分離しようという考え方です。

本章では、コンソールベースの単純なアプリケーションの作成を通じて、Seasar2を紹介します。個々のアプリケーションで利用している機能についての詳細な説明は後続の章で行います。まずはSeasar2を使ったアプリケーションの雰囲気を感じ取ってください。

## 3.2. はじめてのSeasar2アプリケーション

最初に作成するアプリケーションでは、Seasar2を単なるファクトリのように使用します。

### コンポーネントクラス

まずはコンポーネントクラスを作成します。Seasar2におけるコンポーネントは特別なクラスではなく、ごく普通のクラスです。

以下にコンポーネントクラスex01/Hello.javaを示します。

```
package ex01;

public class Hello {
    public String say() {
        return "Hello, Seasar2";
    }
}
```

例 3.1. ex01/Hello.java

say() メソッドが呼び出されると文字列を返すだけの単純なクラスです。

### 設定ファイル

Seasar2を使用するには、使用するコンポーネントの情報を定義した設定ファイルを作成します。Seasar2の標準的な設定ファイルはXML形式で記述します。この設定ファイルの拡張子は 'dicon'（Dependency Injection Configuration）となります。

以下に設定ファイルex01/app.diconを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container 2.4//EN"
"http://www.seasar.org/dtd/components24.dtd">
<components>
  <component name="hello" class="ex01.Hello"/>
</components>
```

- ① 設定ファイルのルート要素は<components>要素です。
- ② コンポーネントの設定を<component>要素で定義しています。<component>要素のname属性でコンポーネント名を、class属性でコンポーネントクラスを指定します。<components>要素には複数の<component>要素を定義することができます。

例 3.2. ex01/app.dicon

### メインクラス

アプリケーションを実行するためのメインクラスex01/Main.javaを作成します。

```
package ex01;
import org.seasar.framework.container.S2Container;
import org.seasar.framework.container.factory.SingletonS2ContainerFactory;

public class Main {
    public static void main(String[] args) {
        SingletonS2ContainerFactory.setConfigPath("ex01/app.dicon");
        SingletonS2ContainerFactory.init();
        S2Container container = SingletonS2ContainerFactory.getContainer();
        Hello hello = (Hello) container.getComponent(Hello.class);
        String message = hello.say();
        System.out.println(message);
    }
}
```

- ❶ 使用する設定ファイルのパスを指定します。
- ❷ コンテナを初期化します。
- ❸ コンテナを取得します。
- ❹ コンテナからHelloコンポーネントを取得します。
- ❺ Helloコンポーネントのメソッドを呼び出し、メッセージを取得します。
- ❻ メッセージを出力します。

例 3.3. ex01/Main.java

## 実行

メインクラスを実行すると、標準出力には次のように出力されます。

```
DEBUG 2007-02-27 20:10:23,562 [main] S2Container#####path=ex01/app.dicon
DEBUG 2007-02-27 20:10:23,750 [main] S2Container#####path=ex01/app.dicon
INFO 2007-02-27 20:10:23,796 [main] Running on [ENV]product, [DEPLOY MODE]Normal Mode
Hello, Seasar2
```

例 3.4. 実行結果

Seasar2によって生成されたHelloクラスのインスタンスを取得し、say()メソッドを呼び出すことができました。

## 3.3. プロパティの設定

次のアプリケーションでは、コンポーネントのプロパティにSeasar2で値を設定します。

### コンポーネントクラス

messageという文字列型のプロパティを持ったコンポーネントクラスex02/Hello.javaを作成します。

```
package ex02;

public class Hello {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String say() {
        return message;
    }
}
```



```
}
}
```

例 3.5. ex02/Hello.java

## 設定ファイル

設定ファイルでGreetingクラスのmessageプロパティにメッセージ文字列を設定します。

以下に設定ファイルex02/app.diconを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container 2.4//EN"
"http://www.seasar.org/dtd/components24.dtd">
<components>
  <component name="hello" class="ex02.Hello">
    <property name="message">"Hello, Dicon!!"</property>
  </component>
</components>
```

- ❶ <property>要素を使ってプロパティの値を設定しています。 name属性でプロパティの名前を、要素の内容でプロパティに設定する値を指定します。

例 3.6. ex02/app.dicon

## メインクラス

アプリケーションを実行するためのメインクラスex02/Main.javaを作成します。

```
package ex02;
import org.seasar.framework.container.S2Container;
import org.seasar.framework.container.factory.SingletonS2ContainerFactory;

public class Main {
    public static void main(String[] args) {
        SingletonS2ContainerFactory.setConfigPath("ex02/app.dicon");
        SingletonS2ContainerFactory.init();
        S2Container container = SingletonS2ContainerFactory.getContainer();
        Hello hello = (Hello) container.getComponent(Hello.class);
        String message = hello.say();
        System.out.println(message);
    }
}
```

例 3.7. ex02/Main.java

## 実行

メインクラスを実行すると、標準出力には次のように出力されます。

```
DEBUG 2007-02-27 19:37:56,031 [main] S2Container#####path=ex02/app.dicon
DEBUG 2007-02-27 19:37:56,375 [main] S2Container#####path=ex02/app.dicon
INFO 2007-02-27 19:37:56,437 [main] Running on [ENV]product, [DEPLOY MODE]Normal Mode
Hello, Dicon!!
```

例 3.8. 実行結果

設定ファイルで指定したメッセージが表示されました。

## 3.4. DIの導入

次のアプリケーションでは、複数のコンポーネントの依存性をSeasar2によって設定します。

### コンポーネントクラス

今回は、コンポーネントクラスを2つ作成します。

まずはコンポーネントクラスex03/Hello.javaを作成します。

```
package ex03;

public class Hello {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String say() {
        return message;
    }
}
```

例 3.9. ex03/Hello.java

続いてコンポーネントクラスex03/Greeting.javaを作成します。

```
package ex03;
public class Greeting {
    Hello hello;

    public Hello getHello() {
        return hello;
    }

    public void setHello(Hello hello) {
        this.hello = hello;
    }

    public void greet() {
        String message = hello.say();
        System.out.println(message);
    }
}
```

例 3.10. ex03/Greeting.java

GreetingクラスはHello型のプロパティを持っています（Helloに依存しています）。

### 設定ファイル

設定ファイルでHelloクラスとGreetingクラスを定義します。

以下に設定ファイルex03/app.diconを示します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container 2.4//EN"
"http://www.seasar.org/dtd/components24.dtd">
<components>
```

```
<component name="hello" class="ex03.Hello">
  <property name="message">"Hello, DI!!"</property>
</component>
<component name="greeting" class="ex03.Greeting">
  <property name="hello">hello</property>
</component>
</components>
```

❶

❷

❸

- ❶ Helloコンポーネントを定義しています。
- ❷ Greetingコンポーネントを定義しています。
- ❸ <property>要素を使ってプロパティの値を設定しています。 GreetingクラスのhelloプロパティにHelloクラスのコンポーネントを設定します。

例 3.11. ex03/app.dicon

## メインクラス

アプリケーションを実行するためのメインクラスex03/Main.javaを作成します。

```
package ex03;
import org.seasar.framework.container.S2Container;
import org.seasar.framework.container.factory.SingletonS2ContainerFactory;

public class Main {
    public static void main(String[] args) {
        SingletonS2ContainerFactory.setConfigPath("ex03/app.dicon");
        SingletonS2ContainerFactory.init();
        S2Container container = SingletonS2ContainerFactory.getContainer();
        Greeting greeting = (Greeting) container.getComponent(Greeting.class);
        greeting.greet();
    }
}
```

例 3.12. ex03/Main.java

## 実行

メインクラスを実行すると、標準出力には次のように出力されます。

```
DEBUG 2007-02-27 20:23:11,187 [main] S2Container#####path=ex03/app.dicon
DEBUG 2007-02-27 20:23:11,406 [main] S2Container#####path=ex03/app.dicon
INFO 2007-02-27 20:23:11,453 [main] Running on [ENV]product, [DEPLOY MODE]Normal Mode
Hello, DI!!
```

例 3.13. 実行結果

GreetingはHelloから取得したメッセージを表示しました。

## 3.5. AOPの適用

次のアプリケーションでは、AOPを使用してコンポーネントにロギングの機能を追加します。

### コンポーネントクラス

今回も、コンポーネントクラスを2つ作成します。

まずはコンポーネントクラスex04/Hello.javaを作成します。

```
package ex04;
```

```
public class Hello {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String say() {
        return message;
    }
}
```

例 3.14. ex04/Hello.java

続いてコンポーネントクラスex04/Greeting.javaを作成します。

```
package ex02;

public class Greeting {
    private String message;

    public String getMessage() {
        return message;
    }

    public void setMessage(String message) {
        this.message = message;
    }

    public String say() {
        return message;
    }
}
```

例 3.15. ex04/Greeting.java

## 設定ファイル

設定ファイルex04/app.diconでGreetingクラスにAOPを適用します。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components PUBLIC "-//SEASAR//DTD S2Container 2.4//EN"
    "http://www.seasar.org/dtd/components24.dtd">
<components>
  <component name="hello" class="ex04.Hello">
    <property name="message">"Hello, AOP!!"</property>
  </component>
  <component name="greeting" class="ex04.Greeting">
    <property name="hello">hello</property>
    <aspect pointcut="greet">traceInterceptor</aspect>
  </component>
  <component name="traceInterceptor"
    class="org.seasar.framework.aop.interceptors.TraceInterceptor"/>
</components>
```

- ❶ <aspect>要素を使ってAOPを設定しています。 pointcut属性でAOPを適用するメソッドを、内容で適用するインターセプタを指定します。
- ❷ Greetingに適用するインターセプタを定義しています。
- ❸ TraceInterceptorクラスはSeasar2が提供するクラスです。

例 3.16. ex04/app.dicon

## メインクラス

アプリケーションを実行するためのメインクラスex04/Main.javaを作成します。

```
package ex04;
import org.seasar.framework.container.S2Container;
import org.seasar.framework.container.factory.SingletonS2ContainerFactory;

public class Main {
    public static void main(String[] args) {
        SingletonS2ContainerFactory.setConfigPath("ex04/app.dicon");
        SingletonS2ContainerFactory.init();
        S2Container container = SingletonS2ContainerFactory.getContainer();
        Greeting greeting = (Greeting) container.getComponent(Greeting.class);
        greeting.greet();
    }
}
```

例 3.17. ex04/Main.java

## 実行

メインクラスを実行すると、標準出力には次のように出力されます。

```
DEBUG 2007-02-27 20:51:52,453 [main] S2Container#####path=ex04/app.dicon
DEBUG 2007-02-27 20:51:52,546 [main] S2Container#####path=aop.dicon
DEBUG 2007-02-27 20:51:52,796 [main] S2Container#####path=aop.dicon
DEBUG 2007-02-27 20:51:52,812 [main] S2Container#####path=ex04/app.dicon
INFO 2007-02-27 20:51:53,203 [main] Running on [ENV]product, [DEPLOY MODE]Normal Mode
DEBUG 2007-02-27 20:51:53,203 [main] BEGIN ex04.Greeting#greet()
Hello, AOP!!
DEBUG 2007-02-27 20:51:53,203 [main] END ex04.Greeting#greet() : null
```

例 3.18. 実行結果

Greetingクラスのgreet()メソッドがメッセージを出力している前後に、AOPによるトレースが出力されています。

## 3.6. まとめ

---

## 第4章 アーキテクチャ

### 4.1.

---

## パート II. S2コンテナ

---

## 第5章 概要

5.1.

5.2.



---

# 第6章 Dependency Injection

## ー依存性の注入ー

### 6.1. はじめに

本章では、Dependency Injection（依存性の注入、以下DI）の利用方法について説明します。

### 6.2. Dependency Injection概要

DIとは、コンポーネントが他のコンポーネントに依存している場合に、その依存物（他のコンポーネント）を外部（S2コンテナ）から注入することです。DIを利用すると、コンポーネントが他のコンポーネントに直接依存することを避けることができます。

図

図

### 6.3. Dependency Injectionのタイプ

Dependency Injectionは、あるコンポーネントが依存する（必要とする）他のコンポーネントをコンテナから渡されます。その際の方法として、次のものがあります。

- コンストラクタ・インジェクション
- セッター・メソッド
- メソッド・インジェクション

#### コンストラクタ・インジェクション

コンストラクタ・インジェクションでは、コンストラクタを通じて依存するコンポーネントが設定されます。

#### セッター・インジェクション

セッター・インジェクションでは、プロパティのセッター・メソッドを通じて依存するコンポーネントが設定されます。

#### メソッド・インジェクション

メソッド・インジェクションでは、任意のメソッドを通じて依存するコンポーネントが設定されます。

メソッド・インジェクション

#### フィールド・インジェクション

フィールド・インジェクションでは、任意のフィールドに直接コンポーネントが設定されます。フィールド・インジェクションはdiconファイルで指定することはできません。フィールド・インジェクションを使用するにはアノテーションを指定します。

### 6.4. ライフサイクルとDI

図が欲しい

- コンストラクタ・インジェクション
- セッター・インジェクション／フィールド・インジェクション
- initメソッド・インジェクション
- 
- destroyメソッド・インジェクション

## 6.5. 自動バインディング

### バインディング・タイプ

`must`

`should` (デフォルト)

`may`

`none`

### 依存コンポーネントの解決

- 型
- 名前

•

### 自動バインディング・モード

`auto`

`constructor`

`property`

`none`

`semiauto` (Version: 2.4～)

## 6.6. 特殊なコンポーネント

### S2Container

### ComponentDef

## 6.7. 外部バインディング

## 6.8. まとめ

---

## 第7章 Aspect-Oriented Programming

ーアスペクト指向プログラミングー

7.1.

7.2. インターセプタの作成

7.3. インタータイプの作成

7.4. 標準インターセプタ

7.5. 標準インタータイプ

---

## 第8章 S2コンテナの作成

8.1.

8.2.

---

## 第9章 S2コンテナの利用

9.1.

9.2.

---

## 第10章 シングルトンS2コンテナ

10.1.

10.2.

---

## 第11章 外部コンテキスト

11.1.

11.2.

---

## パート III. 設定

---



---

## 第12章 概要

12.1.

12.2.

# 第13章 diconファイル

## 13.1. はじめに

本章では、Seasar2の標準的な設定ファイルについて説明します。

Seasar2の標準的な設定ファイルはXML形式のテキストファイルであり、dicon (Dependency Injection Configuration) ファイルと呼ばれます。ファイルの拡張子は 'dicon' です。

## 13.2. 基本構造

diconファイルは次のような構造のXMLファイルです。

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE components
  PUBLIC "-//SEASAR//DTD S2Container 2.4//EN"
  "http://www.seasar.org/dtd/components24.dtd">
<components>
  ...
</components>
```

❶  
❷  
❸  
❹  
❺

- ❶ XML宣言 (任意) : diconファイルの文字エンコーディングがUTF-8以外の場合は必須となります。
- ❷ 文書型宣言 (必須) : Seasar2の定義ファイルであることを示します。
- ❸ パブリックID (必須) : diconファイルのパブリックIDを指定します。
- ❹ システムID (必須) : diconファイルのシステムIDを指定します。
- ❺ <components>要素 (必須) : diconファイルのルート要素です。

diconファイルは妥当 (valid) なXMLファイルでなくてはなりません。

## 13.3. 文書型宣言

diconファイルは妥当 (valid) なXMLファイルでなくてはなりません。そのため、文書型宣言が必須です。文書型宣言で指定するDTDは、使用するSeasar2のバージョンによって、次の中から選択することができます (独自のDTDを使用することも可能です。詳細は「」を参照してください)。

表 13.1. DTDとSeasar2のバージョン

DTDのバージョン	利用可能なSeasar2	パブリックID
		システムID
2.0	Version: 2.0～	-//SEASAR//DTD S2Container//EN
		http://www.seasar.org/dtd/components.dtd
2.1	Version: 2.1～	-//SEASAR2.1//DTD S2Container//EN
		http://www.seasar.org/dtd/components21.dtd
2.3	Version: 2.3～	-//SEASAR//DTD S2Container 2.3//EN
		http://www.seasar.org/dtd/components23.dtd
2.4	Version: 2.4～	-//SEASAR//DTD S2Container 2.4//EN
		http://www.seasar.org/dtd/components24.dtd

DTDによって指定可能な要素や属性およびその値が異なります。 DTDのバージョンとそれぞれで利用可能な要素・属性・値を次の表に示します。

表 13.2. DTDのバージョンと指定可能な要素・属性・値

要素	属性	値	2.0	2.1	2.3	2.4
<components>	(要素自体)	-	○	○	○	○
	namespace	-	○	○	○	○
	initializeOnCreate	-	×	×	×	○
	xmlns:xi	-	×	×	×	○
<description>	(要素自体)	-	○	○	○	○
<include>	(要素自体)	-	○	○	○	○
	path	-	○	○	○	○
	condition	-	×	×	○	○
<xi:include>	(要素自体)	-	×	×	×	○
<component>	(要素自体)	-	○	○	○	○
	name	-	○	○	○	○
	class	-	○	○	○	○
	instance	singleton	○	○	○	○
		prototype	○	○	○	○
		outer	○	○	○	○
		request	×	○	○	○
		session	×	○	○	○
		application	×	×	×	○
	autoBinding	auto	○	○	○	○
		constructor	○	○	○	○
		property	○	○	○	○
		none	○	○	○	○
		semiauto	×	×	×	○
	externalBinding	-	×	×	×	○
<arg>	(要素自体)	-	○	○	○	○
<property>	(要素自体)	-	○	○	○	○
	name	-	○	○	○	○
	bindingType	-	×	×	○	○
<initMethod>	(要素自体)	-	○	○	○	○
	name	-	○	○	○	○

要素	属性	値	2.0	2.1	2.3	2.4
<destroyMethod>	(要素自体)	-	○	○	○	○
	name	-	○	○	○	○
<aspect>	(要素自体)	-	○	○	○	○
	pointcut	-	○	○	○	○
<interType>	(要素自体)	-	×	×	×	○
<meta>	(要素自体)	-	×	○	○	○

## 13.4. <components>要素

ーコンテナ定義ー

<components>要素は、diconファイルのルート要素です。

```
<components
  namespace="nameSpace"
  initializeOnCreate="false | true"
  xmlns:xi="http://www.w3.org/2001/XInclude"
>
  ...
</components>
```

❶  
❷  
❸

- ❶ namespace (任意): このdiconファイルを読み込んだコンテナの名前空間を指定します。
- ❷ initializeOnCreate (任意): コンテナの作成時に初期化も行うことを指定します。 デフォルトはfalseです。
- ❸ xmlns:xi (任意): XIncludeで使用する名前空間を宣言します。

<components>要素の内容には次のものを記述することができます。

- <description>要素 (任意)
- <include>要素 (0個以上)
- <xi:include>要素 (0個以上)
- <component>要素 (0個以上)
- <meta>要素 (0個以上)

<components>要素はルート要素としてのみ記述することができます。他の要素の子となることはできません。

### namespace属性

ー名前空間ー

diconファイルに定義されたコンポーネントの名前空間を指定します。 名前空間の詳細は「」を参照してください。 名前空間を指定すると、コンテナからコンポーネントを名前を取得する際に、名前空間で修飾した名前を指定できるようになります。

#### JavaBeansのプロパティ名

JavaBeansの仕様では、プロパティ名の最初の文字は小文字となります。 ただし、二文字目が大文字の場合は先頭も大文字となります。

正: fooBar, z, URL

誤: FooBar, Z, url, uRL

名前空間は、Javaの識別子として有効な名前を指定します。JavaBeansのプロパティ名と同じルールに従うことを推奨します。

## initializeOnCreate属性

—生成時に初期化—

(Version: 2.4.5～)

コンテナの生成時にコンテナを初期化するかどうかを指定します。

false (デフォルト)

コンテナの生成時には初期化を行いません。

true

コンテナの生成時に初期化を行います。

## xmlns:xi属性

—XInclude名前空間宣言—

(Version: 2.4.0～)

XInclude [http://www.w3.org/TR/xinclude/]を使用する場合は、その名前空間を宣言します()。名前空間接頭辞は 'xi' だけが利用可能です。名前空間URIは 'http://www.w3.org/2001/XInclude' を指定します。この名前空間はXML名前空間仕様 [http://www.w3.org/TR/2006/REC-xml-names-20060816/]のものであり、namespace属性で指定するコンポーネントの名前空間とは別のものです。XIncludeの詳細は「」を参照してください。

## 13.5. <description>要素

—説明—

<description>要素を使用して、diconファイルの説明を記述することができます。

```
<description>
  ...
</description>
```

<description>要素の内容には次のものを記述することができます。

- 任意のテキスト

このテキストはドキュメンテーションのためのものであり、コンテナの動作には影響を与えません。

## 13.6. <meta>要素

—メタ情報—

(Version: 2.1.0～)

コンテナやコンポーネント定義にメタ情報を付けることができます。メタ情報は、コンポーネントをリモートオブジェクトとして公開することを示すなど、様々な用途で使うことができます。

```
<meta
  name="name"
>
  ...
</meta>
```

- ❶ name (任意): メタ情報の名前を指定します。

<meta>要素の内容には次のものを記述することができます。

- 任意のテキスト

- <description>要素 (0個以上)
- <component>要素 (0個以上)
- <meta>要素 (0個以上)

## name属性

—メタ情報名—

メタ情報は名前を持つことができます。 この名前はコンテナやコンポーネント定義からメタ情報を取得する際のキーとして使うことができます。

## 13.7. まとめ

# 第14章 コンポーネント定義

## 14.1. はじめに

本章では、diconファイルでコンポーネントを定義する方法について説明します。

## 14.2. コンポーネント

コンポーネントとは、S2コンテナに管理されるオブジェクトのことです。

S2コンテナで管理可能なコンポーネントは特別なクラスである必要はなく、ほとんどのJavaクラスをコンポーネントとして扱うことが可能です。

## 14.3. <component>要素

ーコンポーネント定義ー

コンポーネントは<component>要素で定義します。

```
<component
  name="componentName"                                ❶
  class="fullQualifiedClassName"                       ❷
  instance="singleton | prototype | outer | request | session | application" ❸
  autoBinding="auto | constructor | property | semiauto | none"             ❹
  externalBinding="true | false"                                           ❺
>
...
</component>
```

- ❶ name (任意): コンポーネントの名前を指定します。
- ❷ class (任意): コンポーネントのクラス名 (完全限定名) を指定します。
- ❸ instance (任意): インスタンス・モードを指定します。デフォルトはsingletonです。
- ❹ autoBinding (任意): 自動バインディング・モードを指定します。デフォルトはautoです。
- ❺ externalBinding (任意): 外部バインディング・モードを指定します。デフォルトはfalseです。

<component>要素の内容には、次のものを記述することができます。

- インスタンス生成式
- <arg>要素
- <property>要素
- <initMethod>要素
- <destroyMethod>要素
- <aspect>要素
- <interType>要素 (Seasar2.4以降)
- <meta>要素 (Seasar2.1以降)
- <description>要素

<component>要素は、次の要素の子として記述することができます。

- <components>要素

- `<arg>`要素
- `<property>`要素
- `<aspect>`要素
- `<interType>`要素 (Seasar2.4以降)
- `<meta>`要素 (Seasar2.1以降)

`<components>`要素以外の要素の子として定義したコンポーネントは、他のコンポーネントから参照することができません。

## name属性

ーコンポーネント名ー

### コンポーネント名

`name`属性で指定するコンポーネント名は、`<components>`要素の直下に定義した`<component>##`のみ有効です。

コンポーネントは名前を持つことができます。この名前は、コンテナからコンポーネントを取得する際のキーとして使うことができます ( )。また、自動バインディングの際にはプロパティ名とのマッチングに使用されます ( )。インスタンス・モードが`request・session・application`の場合、コンポーネント名は必須です。

### JavaBeansのプロパティ名

JavaBeansの仕様では、プロパティ名の最初の文字は小文字となります。ただし、二文字目が大文字の場合は先頭も大文字となります。

正: `fooBar`, `z`, `URL`

誤: `FooBar`, `Z`, `url`, `uRL`

コンポーネント名は、Javaの識別子として有効な名前を指定します。JavaBeansのプロパティ名と同じルールに従うことを推奨します。

コンポーネント名は、`<components>`要素の直下に定義されたコンポーネント定義の場合のみ有効です。`<property>`要素など、他の要素の子として定義された`<component>`要素では、`name`属性は無視されます。

## class属性

ーコンポーネント・クラスー

### ネステッド・クラスを指定する

コンポーネント・クラスにネステッド・クラスを指定するには、外側のクラスとネステッド・クラスの区切りにピリオド (.)ではなく、ドル記号 (\$) を使用します。

`foo.bar.Outer$Inner`

コンポーネントのクラスを指定することができます。この指定の必要性和解釈は`instance`属性の値が`outer`かどうかと、インスタンス生成式の有無によって変わります。

表 14.1. `instance`属性・インスタンス生成式と`class`属性の関係

instance属性	インスタンス生成式	class属性	説明
outer以外	あり	あり	インスタンス生成式によって生成されるインスタンスがコンポーネントとなります。このインスタンスは、 <code>class</code> 属性で指定されたクラスに代入可能でなくてはなりません。代入可能でない場合は、



instance属性	インスタンス生成式	class属性	説明
			<code>org.seasar.framework.container.ClassUnmatchRuntimeException</code> がスローされます。
		なし	インスタンス生成式によって生成されるインスタンスがコンポーネントとなります。
	なし	あり	指定されたクラスのインスタンスがコンポーネントとなります。 インスタンスの生成に使用されるコンストラクタについては、「コンストラクタ・インジェクション」を参照してください。
		なし	エラー。 インスタンス生成式またはclass属性のどちらかが必要です。 両方が省略されると <code>org.seasar.framework.container.factory.TagAttributeNotDefined</code> がスローされます。
outer	-	あり	外部から渡されるインスタンスがコンポーネントとなります。 class属性が指定された場合、外部から渡されるインスタンスは class属性で指定されたクラスに代入可能でなくてはなりません。 代入可能でない場合は、 <code>org.seasar.framework.container.ClassUnmatchRuntimeException</code> がスローされます。
		なし	外部から渡されるインスタンスがコンポーネントとなります。

## instance属性

### ーインスタンス・モード

インスタンス・モードとして、コンポーネントのスコープを指定することができます。

指定可能な値を次に示します。

#### singleton (デフォルト)

コンテナ内で唯一のインスタンスが作成されるモードです。 `singleton`のインスタンスはコンテナの初期化時に作成されます。 コンテナからコンポーネントが取得されると、毎回同一のインスタンスが返されます。

#### prototype

コンテナからコンポーネントが取得される度に、新しいインスタンスが作成されるモードです。

#### outer

外部で作成されたインスタンスがコンテナに渡されるモードです。 コンテナはouterのコンポーネントについてはインスタンスの生成も管理も行いません。 outerは、コンテナの外部で生成されたインスタンスにDIを適用するためのモードです。

#### request (Version: 2.1～)

リクエスト単位にインスタンスが作成されるモードです。 requestモードでは、name属性が必須となります。

requestモードを使用するにはコンテナに外部コンテキストが設定されている必要があります。 外部コンテキストの詳細は「」を参照してください。

Webアプリケーションでは、requestのインスタンスはHttpServletRequestの属性として保持されます。 コンポーネントが取得される際、コンテナはHttpServletRequestからコンポーネント名をキーとして属性を取得します。 属性が存在しなければ、コンポーネントをインスタンス化してHttpServletRequestに設定し、それをコンポーネントとして返します。 属性が存在すれば、それをコンポーネントとして返します ( )。

#### session (Version: 2.1～)

セッション単位にインスタンスが作成されるモードです。 sessionモードでは、name属性が必須となります。

sessionモードを使用するにはコンテナに外部コンテキストが設定されている必要があります。 外部コンテキストの詳細は「」を参照してください。

Webアプリケーションでは、`session`のインスタンスは`HttpSession`の属性として保持されます。コンポーネントが取得される際、コンテナは`HttpSession`からコンポーネント名をキーとして属性を取得します。属性が存在しなければ、コンポーネントをインスタンス化して`HttpSession`に設定し、それをコンポーネントとして返します。属性が存在すれば、それをコンポーネントとして返します。`()`。

`application` (Version: 2.4～)

アプリケーション単位にインスタンスが作成されるモードです。`application`モードでは、`name`属性が必須となります。

`application`モードを使用するにはコンテナに外部コンテキストが設定されている必要があります。外部コンテキストの詳細は「」を参照してください。

Webアプリケーションでは、`application`のインスタンスは`ServletContext`の属性として保持されます。コンポーネントが取得される際、コンテナは`ServletContext`からコンポーネント名をキーとして属性を取得します。属性が存在しなければ、コンポーネントをインスタンス化して`ServletContext`に設定し、それをコンポーネントとして返します。属性が存在すれば、それをコンポーネントとして返します。

## autoBinding属性

ー自動バインディング・モードー

自動バインディングのモードを指定することができます。自動バインディングの詳細は「」を参照してください。

`autoBinding`属性に指定可能な値を次に示します。

`auto` (デフォルト)

コンストラクタ・インジェクションとセッター・インジェクションの両方に自動バインディングが適用されます。

`constructor`

コンストラクタ・インジェクションに対してのみ、自動バインディングが適用されます。

`property`

セッター・インジェクションに対してのみ、自動バインディングが適用されます。

`none`

自動バインディングは適用されません。

`semiauto` (Seasar2.4以降)

アノテーション等で明示的に指定されたプロパティに対してのみ、自動バインディングが適用されます。

## externalBinding属性

ー外部バインディング・モードー

(Version: 2.4.0～)

外部バインディングを使用するかどうかを指定することができます。外部バインディングの詳細は「」を参照してください。

`externalBinding`属性に指定可能な値を次に示します。

`false` (デフォルト)

外部バインディングを使用しません。

`true`

外部バインディングを使用します。

## インスタンス生成式

コンストラクタを使用してインスタンスを生成する場合は「コンストラクタ・インジェクション」を参照してください。

<component>要素の内容に、コンポーネントのインスタンスを生成するためのOGNL式を記述することができます。OGNL式の詳細は「」を参

照してください。

OGNL式を評価した結果のオブジェクトがコンポーネントのインスタンスとなります。

```
<component name="hoge">
  new foo.bar.Hoge()
</component>
```

インスタンス生成式を使うことにより、シングルトン・パターンやファクトリによって取得したインスタンスや定数をコンポーネントとして扱うことができます。

```
<component name="hoge">
  @foo.bar.HogeFactory@getInstance()
</component>
```

インスタンス生成式は、コンポーネントが作成される際に評価されます。

<component>要素にclass属性が指定された場合は、インスタンス生成式を評価した結果のオブジェクトはclass属性で指定されたクラスに代入可能でなくてはなりません。 代入可能でない場合は、`org.seasar.framework.container.ClassUnmatchRuntimeException`がスローされます。

## 14.4. まとめ

# 第15章 DI定義

## ー依存性の注入ー

### 15.1. はじめに

本章では、DI (Dependency Injection, 依存性の注入) の利用方法について説明します。Seasar2が提供するDIの詳細は「[DI](#)」を参照してください。

### 15.2. <arg>要素

#### ーコンストラクタ・インジェクションー

##### <arg>要素の用途

<arg>要素はコンストラクタ・インジェクションだけではなく、メソッド・インジェクションでも使われます。<components>要素の直下の<arg>要素はコンストラクタ・インジェクション、<initMethod>または<destroyMethod>直下の<arg>要素はメソッド・インジェクションとなります。

コンストラクタ・インジェクションは、<component>要素の子として<arg>要素を記述することによって指定します。<arg>要素は、呼び出したいコンストラクタの引数の数だけ記述します。

```
<component class="fullQualifiedClassName">
  <arg>content</arg>
  <arg>content</arg>
  ...
</component>
```

<arg>要素の内容には次のものを記述することができます。

- OGNL式または<component>要素
- <meta>要素 (Seasar2.1以降)
- <description>要素

##### コンストラクタを呼び出せない場合

コンストラクタ呼び出しでコンポーネントを直接インスタンス化できない場合は、<arg>要素を指定せず「インスタンス生成式」を指定してください。

<arg>要素の内容がコンストラクタの引数として渡されます。<arg>要素を複数記述した場合は、その内容が記述した順番にコンストラクタ引数に渡されます。<arg>要素の数とその内容(型)を適用できるコンストラクタが存在しない場合は、`org.seasar.framework.beans.ConstructorNotFoundRuntimeException`がスローされます。

<component>要素にインスタンス生成式(OGNL式)を記述した場合は、<arg>要素が記述されていても無視されます。<arg>要素の内容にOGNL式と<component>要素の両方を記述した場合、<component>要素は無視されます。

### 15.3. <property>要素

#### ーセッター・インジェクションー

```
<component class="fullQualifiedClassName">
```

```

<property
  name="propertyName"
  bindingType="must | should | may | none"
>
  content
</property>
</component>

```

❶  
❷

- ❶ name (必須): プロパティの名前を指定します。
- ❷ bindingType (任意): バインディング・タイプを指定します。デフォルトはshouldです。

### JavaBeansのプロパティ名

JavaBeansの仕様では、プロパティ名の最初の文字は小文字となります。ただし、二文字目が大文字の場合は先頭も大文字となります。

正: fooBar, z, URL

誤: FooBar, Z, url, uRL

<property>要素の内容には次のものを記述することができます。

- OGNL式または<component>要素
- <meta>要素 (Seasar2.1以降)
- <description>要素

<property>要素の内容にOGNL式と<component>要素の両方を記述した場合、<component>要素は無視されます。

## name属性

ープロパティ名ー

## bindingType属性

ーバインディング・タイプー

(Version: 2.3.0～)

must

should (デフォルト)

may

none

## 15.4. <initMethod>要素

ーinitメソッド・インジェクションー

```

<component class="fullQualifiedClassName">
  <initMethod
    name="methodName"
  >
    <arg>content</arg>
    <arg>content</arg>
    ...
  </initMethod>
</component>

```

❶

- ❶ name属性（任意）：メソッドの名前を指定します。

#### <arg>要素について

<arg>要素については「」を参照してください。

<initMethod>要素の内容には次のものを記述することができます。

- OGNL式または<arg>要素
- <description>要素

<initMethod>要素にname属性でメソッド名を指定した場合は、そのメソッドの引数を<arg>要素を記述します。メソッドに引数がない場合は<arg>要素は不要です。

```
<component class="java.util.HashMap">
  <initMethod name="clear"/>
  <initMethod name="put">
    <arg>"key"</arg>
    <arg>"value"</arg>
  </initMethod>
</component>
```

例 15.1. <initMethod>要素にname属性を指定した場合

<initMethod>要素にname属性を指定しなかった場合は、内容としてOGNL式を記述します。OGNL式の中では、`#self`という変数で<initMethod>が記述されているコンポーネントを参照することができます。

```
<component class="java.util.HashMap">
  <initMethod>
    #self.clear()
  </initMethod>
  <initMethod>
    #self.put("key", "value")
  </initMethod>
</component>
```

例 15.2. <initMethod>要素にname属性を指定しなかった場合

<initMethod>要素にname属性が指定された場合は、OGNL式が記述されていても無視されます。

## 15.5. <destroyMethod>要素

ーdestroyメソッド・インジェクションー

#### <destroyMethod>の適用対象

<destroyMethod>は<component>要素のinstance属性がsingletonの場合に限り有効です。

```
<component class="fullQualifiedClassName">
  <destroyMethod
    name="methodName"
  >
    <arg>content</arg>
    <arg>content</arg>
    ...
  </destroyMethod>
</component>
```

❶

- ❶ name属性（任意）：メソッドの名前を指定します。

<arg>要素について

<arg>要素については「」を参照してください。

<destroyMethod>要素の内容には次のものを記述することができます。

- OGNL式または<arg>要素
- <description>要素

<destroyMethod>要素にname属性でメソッド名を指定した場合は、そのメソッドの引数を<arg>要素で記述します。メソッドに引数がない場合は<arg>要素は不要です。

```
<component class="java.util.HashMap">
  <destroyMethod name="remove">
    <arg>"key"</arg>
  </destroyMethod>
</component>
```

例 15.3. <destroyMethod>要素にname属性を指定した場合

<destroyMethod>要素にname属性を指定しなかった場合は、内容としてOGNL式を記述します。OGNL式の中では、#selfという変数で<destroyMethod>が記述されているコンポーネントを参照することができます。

```
<component class="java.util.HashMap">
  <destroyMethod>
    #self.remove("key")
  </destroyMethod>
</component>
```

例 15.4. <destroyMethod>要素にname属性を指定しなかった場合

<destroyMethod>要素にname属性が指定された場合は、OGNL式が記述されていても無視されます。

## 15.6. まとめ

# 第16章 AOP定義

## ーアスペクト指向プログラミングー

### 16.1. はじめに

### 16.2. <aspect>要素

アスペクト

```
<component class="fullQualifiedClassName">
  <aspect
    pointcut="pointcut"
  >
    content
  </aspect>
</component>
```

❶ pointcut (任意): インターセプタを適用するメソッドを選択するためのポイントカットを指定します。

<aspect>要素の内容には次のものを記述することができます。

- OGNL式または<component>要素
- <description>要素

<aspect>要素の内容にOGNL式と<component>要素の両方を記述した場合、<component>要素は無視されます。

<component>要素には複数の<aspect>要素を記述することができます。 その場合、複数の<aspect>要素のpointcut属性で選択されたメソッドには、 複数のインターセプタが<aspect>要素の記述された順番で適用されます。

### pointcut属性

ーポイントカッター

インターセプタを適用するメソッドを正規表現で指定します。 正規表現はカンマ区切りで複数並べることができます。

次の例では、doで始まるメソッドとinitで始まるpublicメソッドが選択されます。

```
<component name="hoge">
  <aspect pointcut="do.*,init.*">traceInterceptor</aspect>
</component>
```

オーバーライドされたメソッドの一部だけを選択することはdiconではできません。 同名のメソッドは全部選択されます (ただしpublicメソッドのみ)。 アノテーションでは特定のメソッドだけを選択することができます。 「」

### 16.3. <interType>要素

ーインタータイプー

(Version: 2.4.0～)

```
<component class="fullQualifiedClassName">
  <interType>
    content
  </interType>
</component>
```



<interType>要素の内容には次のものを記述することができます。

- OGNL式または<component>要素
- <description>要素

<interType>要素の内容にOGNL式と<component>要素の両方を記述した場合、<component>要素は無視されます。

<component>要素には複数の<interType>要素を記述することができます。 その場合、 複数のインタータイプが<interType>要素の記述された順番で適用されます。

## 16.4. まとめ

---

## 第17章 OGNL

17.1.

17.2.

---

## 第18章 アノテーション

18.1.

18.2.

---

## 第19章 diconファイルの分割と編成

19.1.

19.2.

---

## 第20章 SMART deploy

20.1.

20.2.

---

## 第21章 コンポーネントの自動登録

21.1.

21.2.

---

## 第22章 S2コンテナのカスタマイズ

22.1.

22.2.

---

# パート IV. Web

- overview
- S2ContainerServlet
- S2ContainerFilter
- Portlet



---

# パート V. データアクセス

- overview
- transaction management
- declarative transaction
- connection pooling
- jdbc

---

# パート VI.

- overview
- EJB3
- S2Dxo
- standard components
- standard dicons

---

# パート VII. テスト

- [overview](#)
- [S2Unit](#)
- [S2Unit4](#)
- [S2Dataset](#)

---

# パート VIII. for Framework Programmer

- [overview](#)
- [logging](#)
- [S2ContainerBuilder](#)

---

# 索引