Seasar Conference
2006 Autumn

Seasar.NETプロダクトによる Windowsアプリケーション開発

2006.11.12

Seasar.NET
杉本 和也

自己紹介

- 杉本 和也と申します
- 高知県の株式会社アイビスに勤務しています
- 2005/4/29からSeasarで活動しています
  - S2Container.NETとS2Dao.NETのコミッタ
  - Seasar.NETのリーダ

アジェンダ

- Seasar.NETプロダクトのご紹介
- S2Container.NETの使い方
  - DIについて
  - AOPについて
- S2Dao.NETの使い方について
- S2Container.NETとS2Dao.NETを用いたWindowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

Seasar.NETプロダクト紹介

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

Seasar.NETプロダクト紹介

- S2Container.NET
  - AOPをサポートしたDIコンテナ
  - JavaのSeasar2 (S2Container)を.NET Frameworkに移植
- S2Dao.NET
  - O/Rマッピングフレームワーク
  - JavaのS2Daoを.NET Frameworkに移植
- 共に .NET 1.1 と .NET 2.0に対応

Seasar.NETプロダクトの歴史

- JavaのSeasar2とS2Daoの品質と生産性の高さを .NET環境でも！と夢見て開発をスタート

- S2Container.NET
  - 2005年4月29日に開発スタート
  - 現在はバージョン 1.2.6

- S2Dao.NET
  - 2005年9月27日に開発スタート
  - 現在はバージョン 1.0.1

S2Container.NETの使い方

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

- S2Container.NETはDIコンテナ
- DIコンテナを使うと何が良くなるのか？

- アプリケーションの変更が楽になる
- テストが簡単にできるようになる
- 開発の分業が簡単にできるようになる

- **品質・生産性が向上する！**

- 直訳するとDIとは「依存性注入」
- 簡単に説明するとDIコンテナは「必要なコンポーネント (Object) をセットしてくれる頼もしいやつ」
- コンポーネントはDIコンテナに格納されている
- 必要に応じてnew (インスタンス化) してくれる

  それではDIコンテナを使わない場合と使った場合を見てみよう

- ICalcLogic (インターフェース) を実装したCalcLogic (クラス) を呼び出している
- つまり実装クラスであるCalcLogicに依存している
- インターフェースを用意する意味が・・

DIを使わない場合

```
public partial class CalcForm : Form
{

    private void calcButton_Click(object sender, EventArgs e)
    {
        ICalcLogic calcLogic = new CalcLogic();

        int result = calcLogic.Plus(3, 5);
    }
}
```

使用するクラスをnewしている

- ソースを良く見るとインターフェース(ICalcLogic)だけで実装クラス(CalcLogic)が無い！
- つまり実装クラスに依存していない

DIを使った場合

```
public partial class CalcForm : Form
{
    private ICalcLogic _calcLogic;

    public ICalcLogic CalcLogic
    {
        set { _calcLogic = value; }
    }

    private void calcButton_Click(object sender, EventArgs e)
    {
        int result = _calcLogic.Plus(3, 5);
    }
}
```

プロパティを用意しておくとDIコンテナがCalcLogicをnewしてセットしてくれる

DIコンテナへのコンポーネントの登録

- DIコンテナへのコンポーネントの登録はDicon(ダイコン)ファイルで行う

```
<components>

  <!-- 計算ロジック -->
  <component class="CalcLogic" />

  <!-- 計算画面 -->
  <component class="CalcForm" />

</components>
```

Diconファイルをアセンブリに埋め込む場合はビルドアクションプロパティを「埋め込まれたリソース」に設定する。アセンブリに埋め込まずファイルとして配置することもできる

S2Container.NET
DIコンテナからコンポーネントを取り出す

- 以下のようにコンポーネントを取得することができる
- ただし実際は1箇所に記述するだけか、フレームワークが内部的に行いコード中に書かない (ASP.NET:S2HttpModule)

```
public void Main()
{
    // DIコンテナ(S2Container.NET)を作成
    IS2Container container = S2ContainerFactory.Create("App.dicon");

    // DIコンテナを初期化する
    container.Init();

    // CalcFormをDIコンテナから取り出す
    CalcForm calcForm =
        (CalcForm)container.GetComponent(typeof(CalcForm));

    calcForm.Show();
}
```

- プログラム本来の目的以外のコードを外から織り込む

```
public class CalcLogic : ICalcLogic
{
    public int Plus(int x, int y)
    {
        Console.WriteLine("足し算を行います");

        int ret = x + y;

        Console.WriteLine("足し算を行いました");

        return ret;
    }
}
```

本来の目的以外の異なるコード
（こういったコードを埋め込まないようにして、AOPで実現する）

S2Container.NET AOPの設定方法

```
<!-- TraceInterceptor -->
<component name="TraceInterceptor"
    class="Seasar.Framework.Aop.Interceptors.TraceInterceptor" />

 <!-- 計算ロジック -->
<component class="CalcLogic">
  <aspect pointcut="Plus">TraceInterceptor</aspect>
</component>
```

```
BEGIN CalcLogic#Plus(2, 3)

・・・x + y が実行される

END CalcLogic#Plus(2, 3) : 5
```

Plusが呼ばれると開始のログが出力される

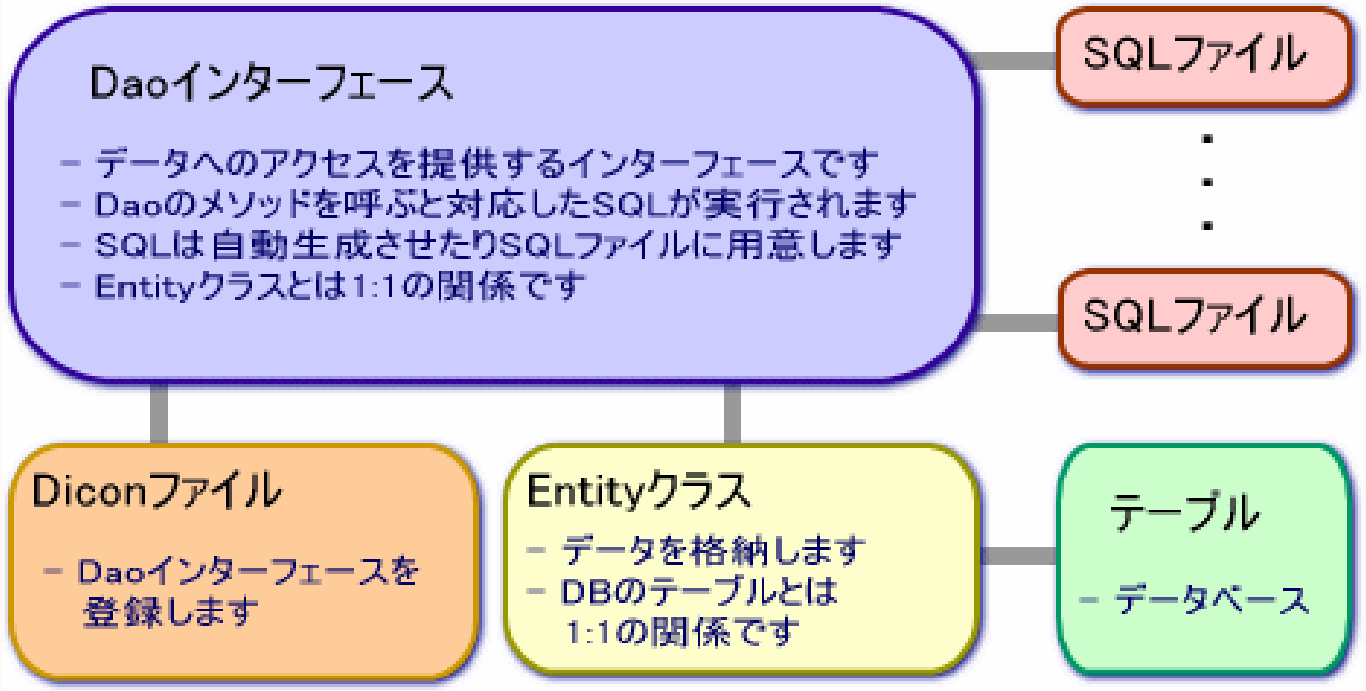
Plusが終わると終了のログが出力される

- 利用ケース
  - ログ出力
  - トランザクション
  - モックの作成 (共同開発時のテストに利用)
  - フレームワークの作成に利用（S2Dao.NET等）
- 利点
  - コードの可読性向上
  - 追加機能のON/OFFをソースコードを変更せずに安全に行える

S2Dao.NETの使い方

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- **S2Dao.NETの使い方**
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

- マッピング情報を設定ファイルに持たないO/Rマッピングフレームワーク
- データアクセス部分の大幅なコストダウン！



テーブルに対応したEntityクラス

- 基本的にテーブルと1対1で作成
- ただしビューのような仮想的なEntityクラスを作成することも可能

Employee（社員テーブル）	
PK	EmpID（社員ID）
U1	EmpCode（社員コード） EmpName（社員名）

テーブル名と同じクラス名（デフォルト）

RDBMSにIDを自動生成してもらう為のID属性

カラム名と同じプロパティ名（デフォルト）

```
public class Employee
{
    private int _empID;
    private int _empCode;
    private string _empName;

    [ID("identity")]
    public int EmpID
    {
        set { _empID = value; }
        get { return _empID; }
    }

    public int EmpCode
    {
        set { _empCode = value; }
        get { return _empCode; }
    }

    public string EmpName
    {
        set { _empName = value; }
        get { return _empName; }
    }
}
```

S2Dao.NET
Entityクラスに指定できる属性

- クラスに指定する属性
  - Table属性 (テーブル名とクラス名が異なる場合に指定)
  - NoPersistentProps属性 (カラムとマッピングしないプロパティを指定)
  - VersionNo属性 (バージョンNoによる排他制御を行うプロパティを指定)
  - Timestamp属性 (タイムスタンプによる排他制御を行うプロパティを指定)
- プロパティに指定する属性
  - Column属性 (カラム名とプロパティ名が異なる場合に指定)
  - RelNo属性, RelKeys属性 (別テーブルとの結合を指定)
  - ID属性 (IDの自動生成を指定)

- Entityクラスと1対1でインターフェースを作成
- 発行するSQLと1対1でメソッドを作成

- 更新系メソッド (メソッド名が下記で始まる)
  - Insert処理 (Insert, Add, Create)
  - Update処理 (Update, Modify, Store)
  - Delete処理 (Delete, Remove)
- 検索系メソッド
  - 更新系メソッド以外で戻り値の型を指定する

- 更新系メソッド
  - SQLを自動生成させる場合、引数はEntityクラス
  - SQLファイルや属性を使ってSQLをカスタマイズ
  - 戻り値の型はSystem.Int32かvoid
    - System.Int32であれば更新行数が戻り値
- 検索系メソッド
  - 引数名からWHERE句を自動生成
  - 戻り値の型がEntityクラスであれば1件分を取得
  - 戻り値の型がEntityクラスの配列, IList, IList<Entityクラス>であれば複数件を取得
  - 戻り値の型が上記以外であれば、1カラムの値を取得

S2Dao.NET
Daoインターフェース

```
[Bean(typeof(Employee))]
public interface IEmployeeDao
{
    void InsertEmp(Employee emp);

    [PersistentProps("EmpName")]
    void UpdateEmp(Employee emp);

    [Sql("delete from Employee where EmpCode=/*empCode*/")]
    void DeleteByEmpCode(int empCode);

    Employee GetByEmpCode(int empCode);

    [Query("order by EmpCode asc")]
    Employee[] GetAllEmployees();

    [Sql("select EmpName from Employee where EmpID=/*empID*/")]
    string GetEmpNameByEmpID(int empID);
}
```

Entityクラスを指定

Insert文を自動生成

更新を行うプロパティを指定

SQLを指定して削除処理

引数名からSELECT文を自動生成
(SELECT・・・ WHERE Empode=3

order by句のみ指定

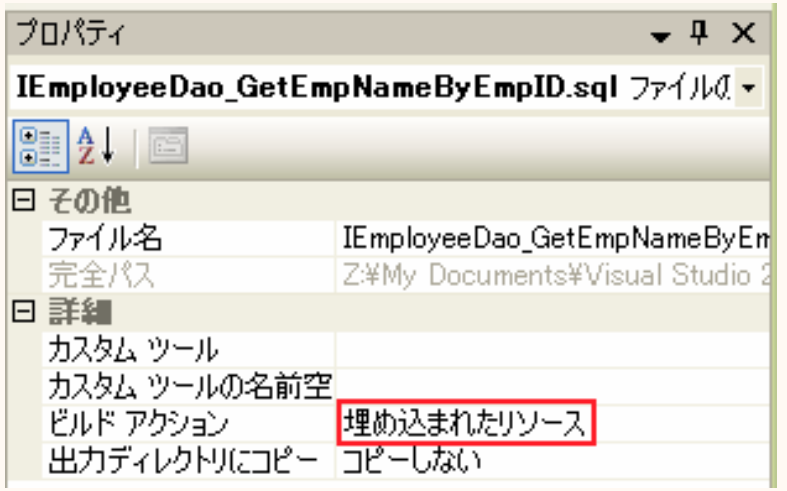
SQLを完全に指定

SQLのコメント記法を使用して引数値をマッピング
(バインド変数コメント)

Daoインターフェースに指定できる属性

- インターフェースに指定
  - Bean属性 (Entityクラスを指定する)
- メソッドに指定
  - Query属性 (Where句以降を指定)
  - Sql属性 (SQLをまるごと指定)
  - NoPersistentProps属性 (自動生成Update文で更新しないプロパティを指定する)
  - PersistentProps属性 (自動生成Update文で更新するプロパティを指定する)

- SQLをまるごと指定できる (SQL属性と同じ)
- Daoインターフェースと同じ名前空間に配置
- ビルドアクションプロパティを「埋め込まれたリソース」に設定
- ファイル名は”*インターフェース名_メソッド名*.sql”



IEmployeeDao_GetEmpNameByEmpID.sql

```
select
      EmpName

from
      Employee

where
      EmpID=/*empID*/3
```

バインド変数コメントの後ろにテスト用のダミーデータ

これによりSQL発行ツールでSQLを実行するとバインド変数コメントの部分が無視され、EmpID=3という値でテストが実行できる

逆にS2Dao.NETで実行する場合は3の部分が無視され引数の値がセットされる

SQLのコメント記法を使用して引数値をマッピング
(バインド変数コメント)

.NETのデータプロバイダを登録する

```
データプロバイダ(OLEDB) -->
<component name="OleDb" class="Seasar.Extension.ADO.DataProvider">
  <property name="ConnectionType">"System.Data.OleDb.OleDbConnection"</property>
  <property name="CommandType">"System.Data.OleDb.OleDbCommand"</property>
  <property name="ParameterType">"System.Data.OleDb.OleDbParameter"</property>
  <property name="DataAdapterType">"System.Data.OleDb.OleDbDataAdapter"</property>
</component>
```

次のページへ続く

前のページからの続き

```
<!-- TransactoinContext（データソースで使用する） -->
<component name="TransactionContext"
           class="Seasar.Extension.Tx.Impl.TransactionContext">
  <property name="IsolationLevel">
      System.Data.IsolationLevel.ReadCommitted
  </property>
</component>

<!-- データソース -->
<component name="SqlDataSource"
           class="Seasar.Extension.Tx.Impl.TxDataSource">
  <property name="DataProvider">OleDb</property>
  <property name="ConnectionString">
    "Provider=Microsoft.Jet.OLEDB.4.0;User ID=admin;Data Source=./sample.mdb"
  </property>
</component>
```

次のページへ続く

DBへの接続文字列と使用するデータプロバイダを指定する

S2Dao.NETのDaoInterceptorとそれに必要なコンポーネントを登録する

前のページからの続き

```
<!-- S2Dao.NETのDaoInterceptorとそれに必要なコンポーネント -->
<component class="Seasar.Extension.ADO.Impl.BasicDataReaderFactory" />
<component class="Seasar.Extension.ADO.Impl.BasicCommandFactory" />
<component class="Seasar.Dao.Impl.DaoMetaDataFactoryImpl" />
<component name="DaoInterceptor"
           class="Seasar.Dao.Interceptors.S2DaoInterceptor"/>

<!-- 社員Dao -->
<component class="IEmployeeDao">
  <aspect>DaoInterceptor</aspect>
</component>
```

IEmployeeDaoにDaoInterceptorを適用する

S2Dao.NET
Daoインターフェースの呼び出し

```
public class EmployeeLogic : IEmployeeLogic
{
    private IEmployeeDao _employeeDao;

    public IEmployeeDao EmployeeDao
    {
        set { _employeeDao = value; }
    }

    public Employee[] GetAllEmployees()
    {
        Employee[] employees = _employeeDao.GetAllEmployees();

        return employees;
    }
}
```

セット用のプロパティを用意してインジェクションしてもらう

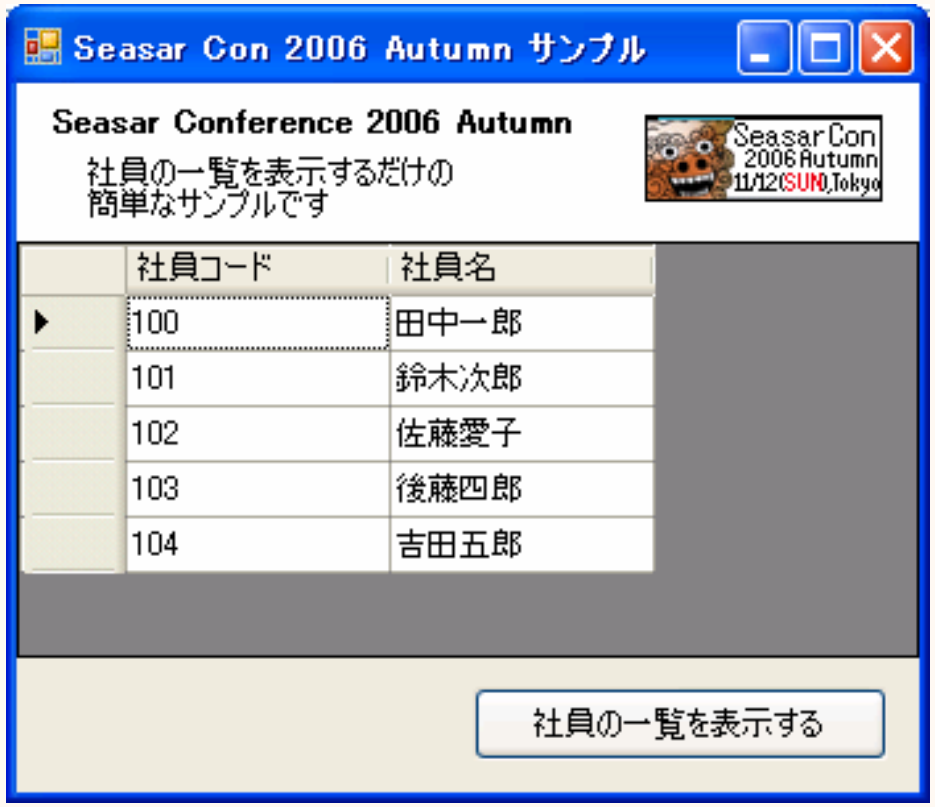
あとはメソッドを呼び出してあげるだけ

ADO.NETのクラス等が出てこない！
ソースコードがすっきり！

Windowsアプリケーション作成

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- **Windowsアプリケーション作成**
- クラス設計について
- Seasar.NETの今後
- まとめ

- ボタンをクリックすると社員の一覧を表示するだけの簡単なサンプル



サンプルプログラムで使用する社員テーブル[Employee]

Windowsアプリケーション作成
ソリューション構成1

処理の呼び出しの流れ

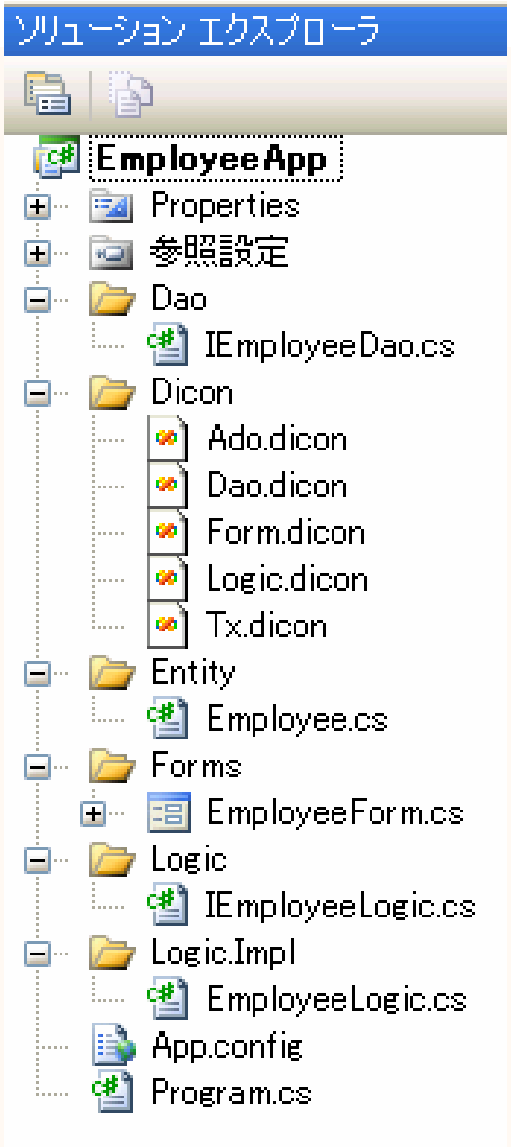
EmployeeForm
(社員表示画面)

↓

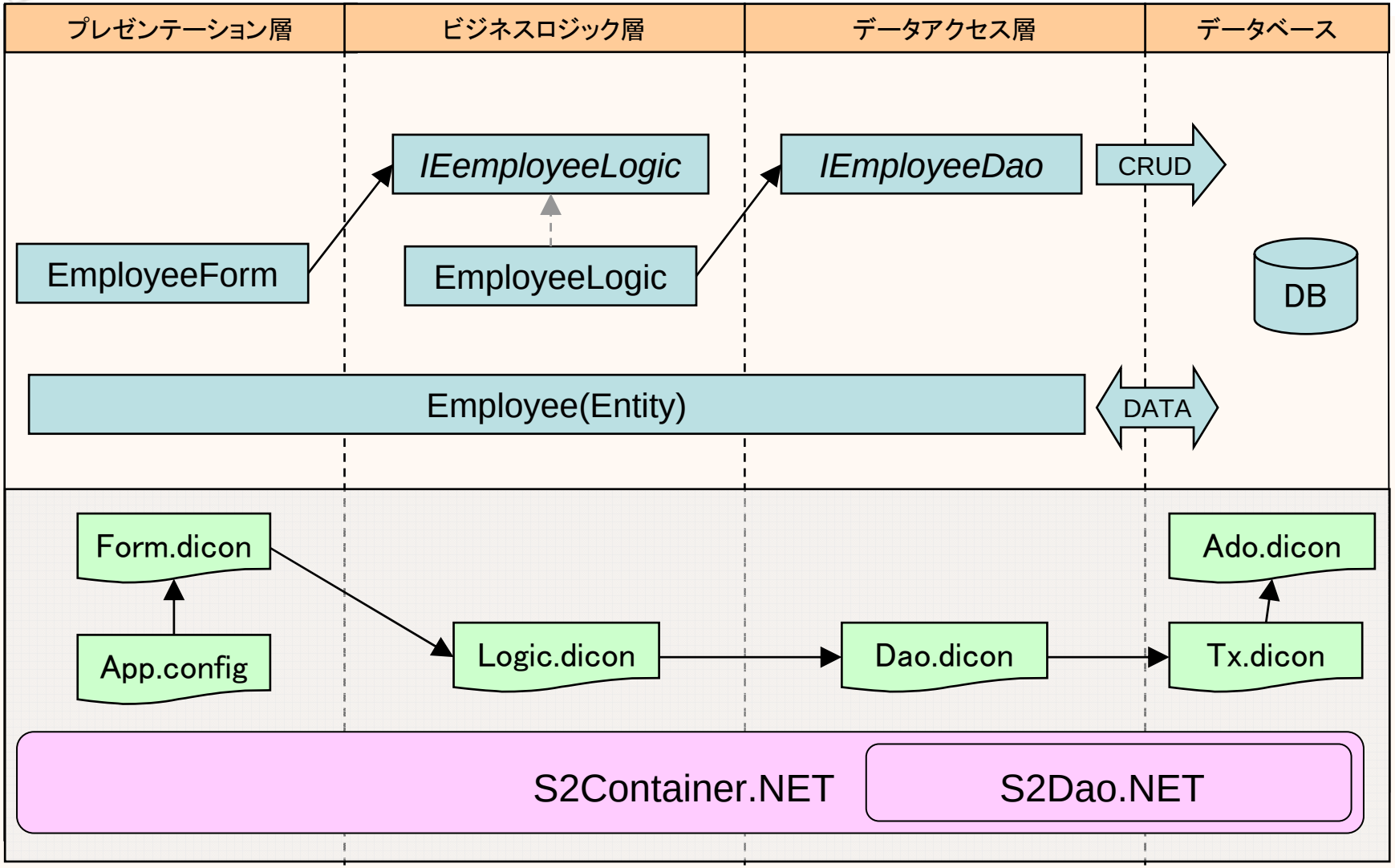
EmployeeLogic
(社員に関するロジック)

↓

IEmployeeDao
(社員に関するDao)

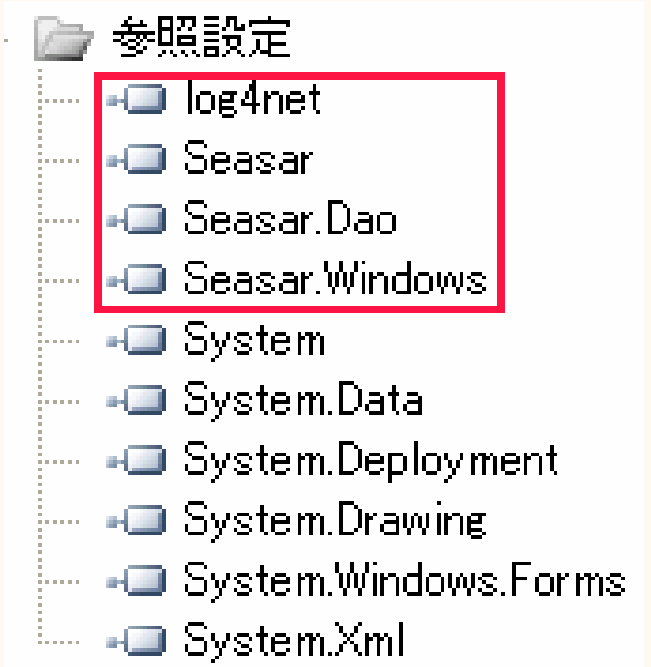


Windowsアプリケーション作成
ソリューション構成2



Windowsアプリケーション作成 参照設定

- 以下のアセンブリをプロジェクトの参照設定に加えます
  - Seasar.dll
    - S2Container.NETの中心となるアセンブリ
  - Seasar.Dao.dll
    - S2Dao.NET
  - Seasar.Windows.dll
    - S2Container.NETに含まれるS2Windows.NET
  - log4net.dll
    - Seasar.NETプロダクト中で使用されているロギングフレームワーク



Windowsアプリケーション作成
アプリケーション構成ファイル(App.config)

```
<?xml version="1.0" encoding="utf-8" ?>
<configuration>

  <configSections>
    <section name="log4net" type="System.Configuration.IgnoreSectionHandler" />
    <section name="seasar" type="Seasar.Framework.Xml.S2SectionHandler, Seasar" />
  </configSections>

  <log4net>
    <appender name="ConsoleAppender" type="log4net.Appender.ConsoleAppender">
      <layout type="log4net.Layout.PatternLayout">
        <conversionPattern value="%-5p %d [%t] %m%n" />
      </layout>
    </appender>
    <root>
      <level value="DEBUG" />
      <appender-ref ref="ConsoleAppender" />
    </root>
  </log4net>
```

log4netとseasarのセクションを宣言する

log4netの設定（コンソールにDEBUGレベル以上を出力）

次のページへ続く

Windowsアプリケーション作成
アプリケーション構成ファイル(App.config)

ルートとなるDiconファイルを指定

前のページからの続き

```
<seasar>
    <configPath>EmployeeApp.Dicon.Form.dicon</configPath>
    <assemblys>
      <assembly>Seasar.Dao</assembly>
      <assembly>Seasar.Windows</assembly>
    </assemblys>
  </seasar>

</configuration>
```

コンテナに登録するのに必要なアセンブリを設定する
(コンテナがアセンブリ中のクラスを探せるように。スタートアッププロジェクトのアセンブリは設定しなくて良い)

Windowsアプリケーション作成 Entityクラスの作成

- S2Dao.NETが取得したデータを格納するEntityクラスを作成する
  - テーブル名に対応したEmployeeクラスを作成する
  - カラムに対応した各プロパティを用意する

```
public class Employee
{
    private int _empID;
    private int _empCode;
    private string _empName;

    [ID("identity")]
    public int EmpID
    {
        set { _empID = value; }
        get { return _empID; }
    }

    public int EmpCode
    {
        set { _empCode = value; }
        get { return _empCode; }
    }

    public string EmpName
    {
        set { _empName = value; }
        get { return _empName; }
    }
}
```

- Daoインターフェースと社員の一覧を取得する為のメソッドを用意する

```
/// <summary>
/// Employeeテーブルにアクセスする為のDao
/// </summary>
[Bean(typeof(Employee))]
public interface IEmployeeDao
{
    /// <summary>
    /// 社員の一覧を社員コードの昇順で取得する
    /// </summary>
    /// <returns>社員の一覧</returns>
    [Query("order by EmpCode asc")]
    Employee[] GetAllEmployees();
}
```

Windowsアプリケーション作成 Ado.diconの作成

- .NET Data Providerに何を使うかを設定する
- ここを変更するだけで .NET Data Providerを切り替えることができる

```
<components namespace="Ado">

  <!-- データプロバイダ(OLEDB) -->
  <component name="OleDb" class="Seasar.Extension.ADO.DataProvider">
    <property name="ConnectionType">"System.Data.OleDb.OleDbConnection"</property>
    <property name="CommandType">"System.Data.OleDb.OleDbCommand"</property>
    <property name="ParameterType">"System.Data.OleDb.OleDbParameter"</property>
    <property name="DataAdapterType">"System.Data.OleDb.OleDbDataAdapter"</property>
  </component>

</components>
```

Windowsアプリケーション作成 Tx.diconの作成

```
<components>

  <!-- Ado.dicon -->
  <include path="EmployeeApp/Dicon/Ado.dicon" />

  <!-- TransactoinContext（データソースで使用する） -->
  <component name="TransactionContext"
             class="Seasar.Extension.Tx.Impl.TransactionContext">
    <property name="IsolationLevel">
        System.Data.IsolationLevel.ReadCommitted
    </property>
  </component>

  <!-- データソース -->
  <component name="SqlDataSource"
             class="Seasar.Extension.Tx.Impl.TxDataSource">
    <property name="DataProvider">Ado.OleDb</property>
    <property name="ConnectionString">
      "Provider=Microsoft.Jet.OLEDB.4.0;User ID=admin;Data Source=./sample.mdb"
    </property>
  </component>

</components>
```

Ado.diconを読み込む

DBへの接続文字列を設定する

Windowsアプリケーション作成 Dao.diconの作成

```
<components>

  <!-- Tx.dicon -->
  <include path="EmployeeApp/Dicon/Tx.dicon" />

  <!-- S2Dao.NETのDaoInterceptorとそれに必要なコンポーネント -->
  <component class="Seasar.Extension.ADO.Impl.BasicDataReaderFactory" />
  <component class="Seasar.Extension.ADO.Impl.BasicCommandFactory" />
  <component class="Seasar.Dao.Impl.DaoMetaDataFactoryImpl" />
  <component name="DaoInterceptor"
             class="Seasar.Dao.Interceptors.S2DaoInterceptor"/>

  <!-- 社員Dao -->
  <component class="EmployeeApp.Dao.IEmployeeDao">
    <aspect>DaoInterceptor</aspect>
  </component>

</components>
```

Tx.diconを読み込む

- コンポーネントはインタフェースを経由して利用するようにしたい為インターフェースを用意する

```
/// <summary>
/// 社員を扱うLogic
/// </summary>
public interface IEmployeeLogic
{
    /// <summary>
    /// 社員の一覧を取得する
    /// </summary>
    /// <returns>社員の一覧</returns>
    Employee[] GetAllEmployees();
}
```

Windowsアプリケーション作成
Logic実装クラスの作成

```
public class EmployeeLogic : IEmployeeLogic
{
    private IEmployeeDao _employeeDao;

    public IEmployeeDao EmployeeDao
    {
        set { _employeeDao = value; }
    }

    public Employee[] GetAllEmployees()
    {
        // 社員の一覧を取得する
        Employee[] employees = _employeeDao.GetAllEmployees();

        // 社員の一覧を返す
        return employees;
    }
}
```

社員テーブルからデータを取得する為にIEmployeeDao型の設定用プロパティを用意するとS2Container.NETがインジェクションしてくれる

社員Daoを呼び出して、社員の一覧を取得する

ADO.NETのクラス等が出てこない！
ソースコードがすっきり！

- 社員Logicを登録してログを出力する為のTraceInterceptorを適用する

Dao.diconを読み込む

```
<components>

  <!-- Dao.dicon -->
  <include path="EmployeeApp/Dicon/Dao.dicon" />

  <!-- ログを出力する為のTraceInterceptor -->
  <component name="traceInterceptor"
      class="Seasar.Framework.Aop.Interceptors.TraceInterceptor" />

  <!-- 社員Logic -->
  <component class="EmployeeApp.Logic.Impl.EmployeeLogic">
    <aspect pointcut="GetAllEmployees">traceInterceptor</aspect>
  </component>

</components>
```

Windowsアプリケーション作成
EmployeeFormデザインの作成

Employeeクラスから作成したデータソースを下にDataGridViewを配置する

Employeeクラスとバインディングする為のBindingSource

Windowsアプリケーション作成 EmployeeFormの作成

```
public partial class EmployeeForm : Form
{
    private IEmployeeLogic _employeeLogic;

    public IEmployeeLogic EmployeeLogic
    {
        set { _employeeLogic = value; }
    }

    // コンストラクタ省略

    /// [社員の一覧を表示する]ボタン クリック
    private void getEmpButton_Click(object sender, EventArgs e)
    {
        // 社員の一覧を取得する
        Employee[] employees = _employeeLogic.GetAllEmployees();

        // 取得した社員の一覧を表示する
        employeeBindingSource.DataSource = employees;
    }
}
```

社員ロジックを利用する為にIEmployeeLogic型の設定用プロパティを用意するとS2Container.NETが実装クラスをコンテナの中から探し出しインジェクションしてくれる

社員ロジックを利用して社員の一覧を取得する

- S2Windows.NETのS2ApplicationContextを利用して最初に開くFormを制御する

```
<components>

  <!-- Logic.dicon -->
  <include path="EmployeeApp/Dicon/Logic.dicon" />

  <!-- S2ApplicationContext -->
  <component class="Seasar.Windows.S2ApplicationContext" >
    <property name="MainForm">EmployeeForm</property>
  </component>

  <!-- 社員一覧Form -->
  <component name="EmployeeForm"
                        class="EmployeeApp.Forms.EmployeeForm" />

</components>
```

Logic.diconを読み込む

Windowsアプリケーション作成
メインエントリポイントの作成

```
static class Program
{
/// アプリケーションのメイン エントリ ポイントです。
    [STAThread]
    static void Main()
    {
        InitApplication();

        // SingoletonS2ContainerFactoryを初期化する(S2コンテナが作成される)
        SingletonS2ContainerFactory.Init();

        // S2コンテナを取得する
        IS2Container container = SingletonS2ContainerFactory.Container;

        // アプリケーションを実行する
        Application.Run((ApplicationContext)
            container.GetComponent(typeof(S2ApplicationContext)));
    }
```

次のページへ続く

Windowsアプリケーション作成
メインエントリポイントの作成

前のページからの続き

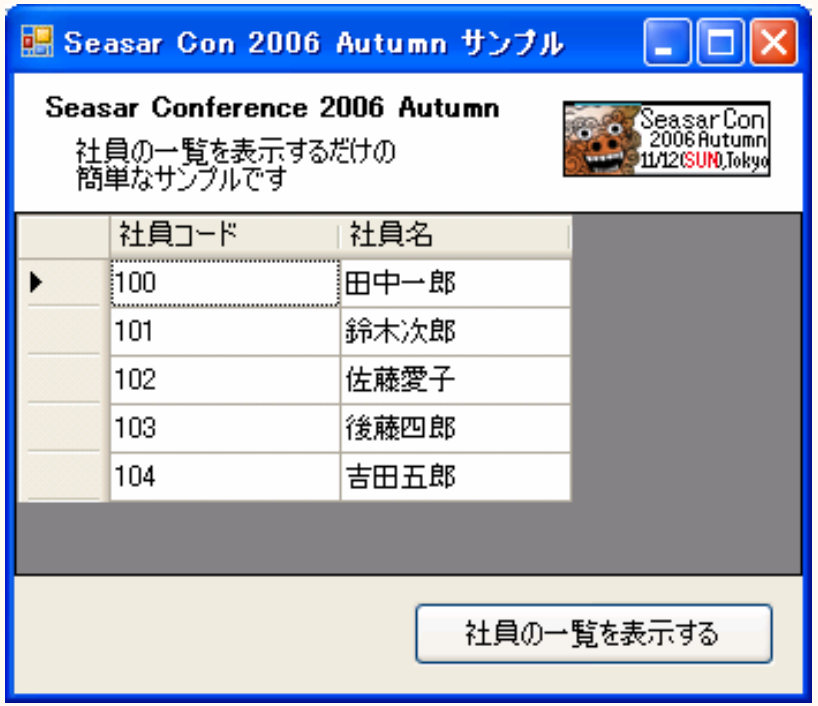
log4netに設定を読ませる

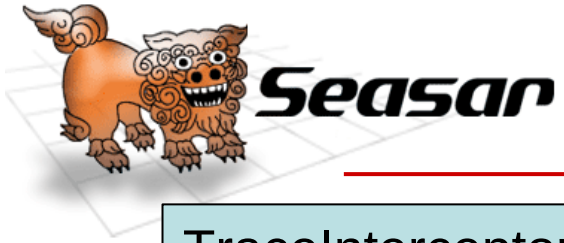
```
private static void InitApplication()
    {
        FileInfo info = new FileInfo(SystemInfo.AssemblyShortName(
          Assembly.GetExecutingAssembly()) + ".exe.config");

        XmlConfigurator.Configure(LogManager.GetRepository(), info);

        Application.EnableVisualStyles();
        Application.SetCompatibleTextRenderingDefault(false);
    }
}
```

- 以上で完成です
- アプリケーションを実行しボタンをクリックすると社員の一覧が表示され、ログが出力されます





Windowsアプリケーション作成 出力されたログ

TraceInterceptorによるログ

社員テーブルのメタ情報取得

```
DEBUG 2006-10-24 20:50:18,897 [10] BEGIN
EmployeeApp.Logic.IEmployeeLogic#GetAllEmployees()

DEBUG 2006-10-24 20:50:19,116 [10] 論理的なコネクションを取得しました
DEBUG 2006-10-24 20:50:19,178 [10] 論理的なコネクションを閉じました

DEBUG 2006-10-24 20:50:19,210 [10] SELECT Employee.EmpName,
Employee.EmpCode, Employee.EmpID FROM Employee order by EmpCode
asc
DEBUG 2006-10-24 20:50:19,225 [10] 論理的なコネクションを取得しました
DEBUG 2006-10-24 20:50:19,288 [10] 論理的なコネクションを閉じました

DEBUG 2006-10-24 20:50:19,288 [10] END
EmployeeApp.Logic.IEmployeeLogic#GetAllEmployees() :
EmployeeApp.Entity.Employee[]
```

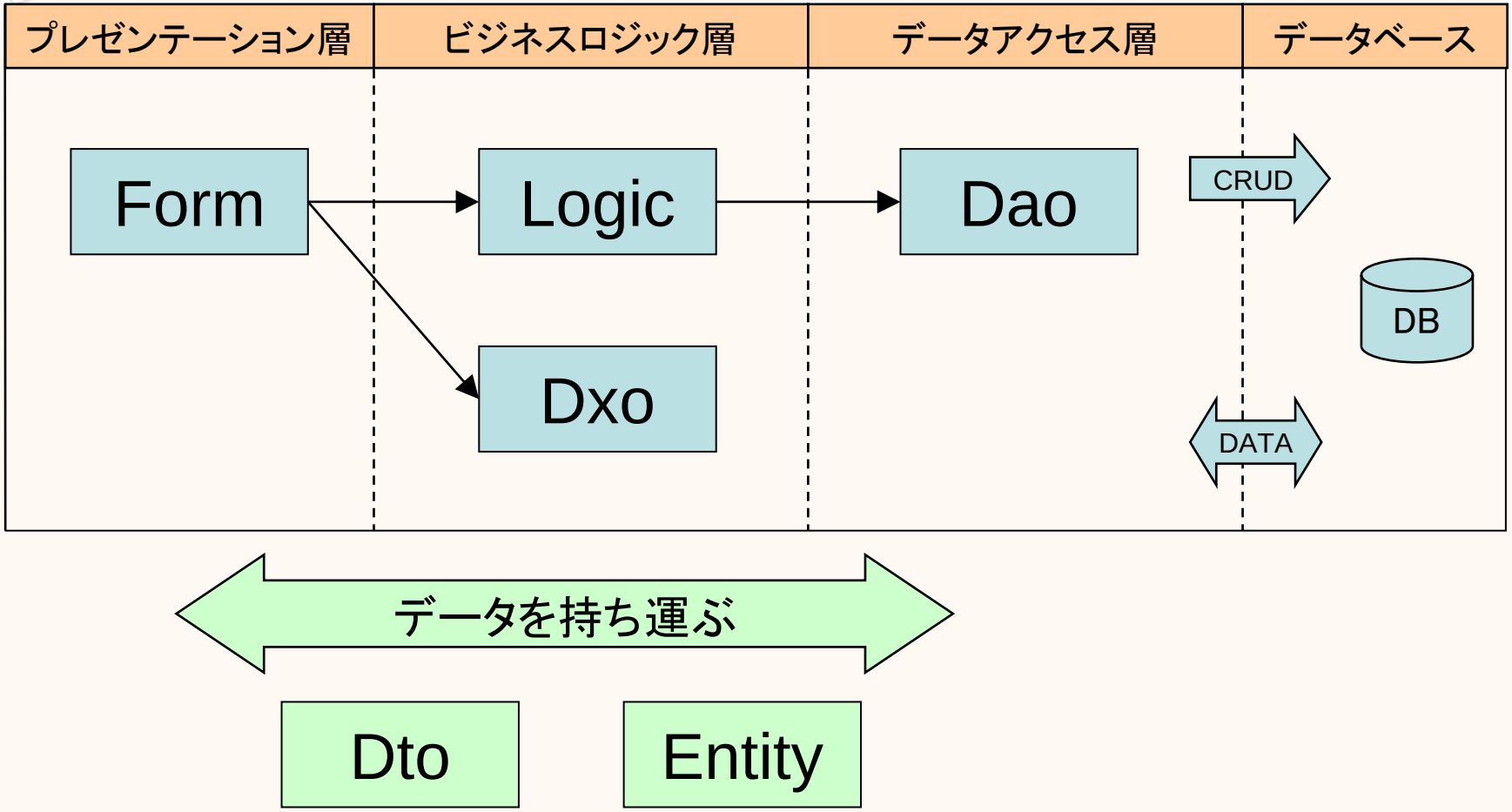
社員の一覧を取得

TraceInterceptorによるログ

クラス設計について

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

クラス設計について
レイヤーアプローチ



- Form
  - System.Windows.Forms.Formクラスを継承したクラス
  - Logicを呼び出し必要な処理を行う
  - プレゼンテーション層とビジネスロジック層でデータの変換が必要な場合はDxoで変換する
  - ユーザコントロールからLogicやDxoを呼び出しても良い
  - LogicやDxoはインターフェース経由で呼び出す

クラス設計について ビジネスロジック層

- Logic
  - Logicインターフェースとそれを実装するLogicクラス
  - 処理を行う軸となるデータごとに作成する
- Dxo（Data eXchange Object）
  - プレゼンテーション層とビジネスロジック層の間でデータの変換が必要な場合は、ここでデータの変換を行う
- Logic, Dxo共にステートレスに実装する
  - クラス変数に状態を持たない

クラス設計について ステートレスに実装1

```
public class EmployeeLogic : IEmployeeLogic
{
    // プロパティの記述は省略
    private int _empID;
    private Employee _employee;

    public void GetEmpByEmpID()
    {
        // 社員IDを条件に社員を取得する
        _employee = _employeeDao.GetByEmpID(_empID);
    }
}
```

条件や結果にクラス変数を使用している

ステートレスに実装すると下のようになる

```
public class EmployeeLogic : IEmployeeLogic
{
    public Employee GetEmpByEmpID(int empID)
    {
        // 社員IDを条件に社員を取得する
        return _employeeDao.GetByEmpID(empID);
    }
}
```

条件は引数に、結果は戻り値を使用している

クラス設計について
ステートレスな実装の利点

- ステートレスな場合（状態を持たない）
  - 引数に始まり、戻り値に終わるので、単体テストがシンプル
- ステートフルな場合（状態を持つ）
  - クラスが肥大化した場合にクラス変数がどのメソッドでどのように使われているか管理が大変
    - しっかりとした技術力が必要となり複数人での管理も大変
- なぜ静的クラス（static class）ではないか
  - テストや共同作業での利点を考え、インターフェース経由で利用したいから

- Dao（Data Access Object）
  - Daoインターフェースとそれを実装するDaoクラス（ただしS2Dao.NETを利用する場合はインターフェースのみ）
  - データベースのテーブルやビュー毎に作成する
  - Entityクラスとも1対1の関係

- Entity
  - データベースのテーブルやビュー毎に作成するクラス
  - 1レコード分のデータを持ち運ぶ
- Dtoクラス（Data Transfer Object）
  - Entity以外のデータを持ち運ぶクラス
- Interceptor
  - Aspectで処理を注入する為のクラス
- Util
  - Utilインターフェースとそれを実装するUtilクラス
  - 特定のプロトコルやプログラム言語に依存する処理を扱う

Seasar.NETの今後

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

Seasar.NETの今後

- S2Container.NET 2.0の開発
  - Seasar 2.4ベースの新しい設計
- S2Dao.NET 1.5の開発
  - S2Pager, データセット対応
- S2Remoting.NETのSandbox卒業
  - リモートオブジェクトを簡単に
- ドキュメントとサンプルの充実

まとめ

- Seasar.NETプロダクト紹介
- S2Container.NETの使い方
- S2Dao.NETの使い方
- Windowsアプリケーション作成
- クラス設計について
- Seasar.NETの今後
- まとめ

まとめ

- S2Container.NET
  - 実装クラスに依存せずインターフェース経由でやりとりを行える
  - 変更、テスト、分業が行いやすい
- S2Dao.NET
  - マッピング情報をXMLに持たないので簡単に扱えることができ、劇的に生産性が向上する
  - マッピングミスやADO.NETのAPIの扱いのバグが無くなり品質が向上する
- 問題点
  - Windowsアプリケーションの場合は起動に時間がかかると問題なので、S2Containerの初期化を行わない等、インスタンスの生成を遅らせる工夫が必要

ご清聴ありがとうございました

- S2Container.NET
  - http://s2container.net.seasar.org/
- S2Dao.NET
  - http://s2dao.net.seasar.org/

- sugimotokazuyaの日記
  - http://d.hatena.ne.jp/sugimotokazuya/