Criando um conector de gateway de dados

Para executar o gateway de dados, você deve ter um conector. O Archer tem um conector disponível para uso integrado, denominado Conector SQL Flexível. Para obter mais informações sobre esse conector, consulte Usando o Conector SQL flexível. Você também pode desenvolver seu próprio conector. Use as tarefas a seguir para criar um conector de gateway de dados que leia, grave e pesquise dados armazenados em uma tabela de banco de dados externo. Essas instruções presumem que você já tenha experiência em criar e implementar projetos C# .NET no Microsoft Visual Studio.

O conector lê os valores dos campos em cada registro do aplicativo de gateway de dados no Archer em um banco de dados externo. No exemplo neste tópico, o banco de dados externo inclui uma tabela Registros com 1 coluna para cada campo do aplicativo de gateway de dados. Uma coluna na tabela deve ser um identificador, que associa cada registro da tabela externa ao respectivo registro no aplicativo do gateway de dados. Neste exemplo, o identificador é ID.

Nesta página

Configuração inicial

Tarefa 1: Criar o projeto

Implemente um conector de gateway de dados como uma DLL .NET. Faça o seguinte para criar o projeto.

1. Usando o Microsoft Visual Studio, crie uma nova biblioteca de classe Visual C# com as seguintes configurações:

    - Nome da montagem: ArcherTech.Datasource.SqlDemo

      Observação: todos os nomes de conector do gateway de dados devem começar com o prefixo ArcherTech.Datasource.

    - Estrutura de destino: .NET Framework 4.7.2
2. Adicione as seguintes referências ao projeto:

    - ArcherTech.Datasource.Common (local do conjunto do Archer "C:\inetpub\wwwroot\RSAarcher\bin\ArcherTech.Datasource.Common.dll")

- ◦ ArcherTech.IOC (local do conjunto do Archer "C:\inetpub\wwwroot\RSAarcher\bin\ArcherTech.IOC.dll")
- ◦ System.Runtime.Serialization (parte do .NET Framework)

Tarefa 2: Criar a classe Datasource

A classe Datasource do gateway de dados representa a lógica de programação de um conector. A classe Datasource inclui métodos para inicializar o conector, gerar e analisar identidades de registro externo e devolver objetos dedicados a leitura, gravação e pesquisa de dados. Faça o seguinte para criar a classe Datasource:

1. Adicione ao projeto um novo arquivo de classe chamado SqlDatasource.
2. Atualize a classe para implementar a interface ArcherTech.Datasource.Common.Interfaces.IContentDatasource, deixando a implementação padrão.

Inicialização e configuração da fonte de fonte

O sistema do gateway de dados faz o seguinte para configurar e inicializar um conector:

- Lê os dados de configuração armazenados do conector
- Obtém a instância do modelo de configuração do conector
- Atualiza o modelo de configuração com os dados de configuração
- Passa o modelo de configuração para o método `Initialize()` do conector

Tarefa 3: Especificar o modelo de configuração

O modelo de configuração é uma implementação da interface IDatasourceConfiguration que fornece o sistema do gateway de dados com um conjunto estruturado de instâncias de propriedade nomeadas e vazias que o sistema atualiza com valores de configuração correspondentes. A estrutura do gateway de dados inclui a classe DatasourceConfiguration, que especifica os parâmetros de configuração básicos. Se a implementação exigir lógica de configuração avançada, vá

para a Tarefa 4. Caso contrário, faça o seguinte para especificar o modelo de configuração padrão.

1. Abra a classe SqlDatasource criada na Tarefa 2.
2. Adicione o seguinte código ao método `GetConfigurationTemplate()`:

```
var config = new DatasourceConfiguration();
config.AddConfigurationData<string>("ConnectionString");
return config;
```

Este código retorna a classe DatasourceConfiguration com a propriedade ConnectionString quando o sistema solicita um modelo de configuração. Se a implementação exigir lógica de configuração avançada, consulte a Tarefa 4 para criar um modelo personalizado. Caso contrário, prossiga para Tarefa 5.

Tarefa 4 (opcional): Criar um modelo de configuração personalizado

Você pode criar uma implementação personalizada de IDatasourceConfiguration se sua implementação exigir uma lógica de configuração avançada. Faça o seguinte para criar um modelo de configuração personalizado.

1. Adicione ao projeto um novo arquivo de classe chamado TextFileConfiguration.

2. Implemente a interface ArcherTech.Datasource.Common.Interfaces.IDatasourceConfiguration com o seguinte código:

```
publicclassTextFileConfiguration: IDatasourceConfiguration
{
publicTextFileConfiguration()
{
Properties = newList<DatasourceConfigurationData>
{
newDatasourceConfigurationData
{
Name = "FilePath",
Type = typeof(string)
}
};
}
publicIEnumerable<DatasourceConfigurationData> Properties
{ get; }
```

```
}
```
Este código implementa a propriedade de configuração "Properties" e adiciona uma propriedade para o caminho do arquivo de texto.

3. Abra a classe TextFileConfiguration criada na etapa 1 e localize o método `GetConfigurationTemplate()`.
4. Exclua a seguinte linha de `GetConfigurationTemplate()`:
   ```
   throw new System.NotImplementedException();
   ```
5. Adicione a seguinte linha a `GetConfigurationTemplate()`:
   ```
   return new TextFileConfiguration();
   ```

Este código retorna a classe personalizada TextFileConfiguration com a propriedade FilePath quando o sistema solicita um modelo de configuração.

Tarefa 5: Implementar a inicialização de Datasource

A interface IContentDatasource inclui um método `Initialize(IDatasourceConfiguration)` para a inicialização de Datasource. A API de inicialização é extensível o suficiente para gerenciar qualquer cenário de inicialização personalizado. Você pode usar esse método para passar informações de login para outro sistema, configurações de tempo de espera excedido para solicitações da Web e outros parâmetros. O método Initialize transmite a string de conexão do banco de dados para o conector SQL. Faça o seguinte para implementar a inicialização de Datasource.

1. Exclua a linha padrão:
   ```
   throw new System.NotImplementedException();
   ```

2. Adicione uma importação de namespace na classe:
   ```
   using ArcherTech.Datasource.Common.Extensions;
   ```

3. Adicione um campo privado chamado ConnectionString à classe:
   ```
   private string connectionString;
   ```

4. Adicione o seguinte código dentro do método:
   ```
   public void Initialize(IDatasourceConfiguration
   configuration)
   {
   connectionString =
   configuration.GetPropertyValue("ConnectionString");
   }
   ```

Este código tenta localizar a propriedade DatasourceConfiguration com o nome ConnectionString e definir seu valor como um membro de classe local. A string de conexão é exigida durante a leitura, a gravação e a pesquisa de dados.

Identificando registros

O sistema do gateway de dados representa uma parte lógica do conteúdo montando registros e seus dados de campo de diferentes sistemas. Dentro do contexto do sistema do gateway de dados, cada registro é considerado uma parte do conteúdo. O conector do gateway de dados deve fornecer uma classe ContentPartIdentityBase que contenha informações para identificar corretamente um registro externo no sistema do gateway. O conector analisa e retorna instâncias de identidade da parte do conteúdo durante a atualização do conteúdo e gera novas instâncias ao criar conteúdo. A estrutura do gateway de dados fornece a classe ContentPartIdentityBase como implementação padrão para essa funcionalidade. Essa classe atende aos requisitos básicos de uma classe ContentPartIdentityBase dentro de um conector do gateway de dados.

Opcionalmente, você pode criar uma implementação personalizada da classe ContentPartIdentityBase se sua implementação exigir uma lógica de comparação personalizada.

Realize as tarefas a seguir para permitir que seu conector identifique registros e gere e analise identidades de partes de conteúdo.

Tarefa 6 (opcional): Criar uma classe ContentPartIdentity personalizada

Se a implementação exige uma lógica de comparação personalizada para identificar registros, faça o seguinte para criar uma implementação personalizada da classe ContentPartIdentityBase.

1. Adicione ao projeto uma nova classe chamada TextFileContentPartIdentity.

2. Implemente a classe ArcherTech.Datasource.Common.ContentPartIdentityBase com o seguinte código:

```
[Serializable]
[DataContract]
public class TextFileContentPartIdentity :
ContentPartIdentityBase, ISerializable
{
private readonly string _id;
private const string ID_INFO = "id";
#region Public Constructor
public TextFileContentPartIdentity(string id)
{
_id = id;
}
public TextFileContentPartIdentity(SerializationInfo
```

```
serializationInfo, StreamingContext streamingContext
{
_id = serializationInfo.GetString(ID_INFO);
}
#endregion
#region Public Properties
public string Value => _id;
#endregion
#region Public Methods
public override int CompareTo(object obj)
{
if (obj is TextFileContentPartIdentity)
{
var obj1 = obj as TextFileContentPartIdentity;
return _id.CompareTo(obj1._id);
}
if (obj is string)
{
var obj1 = (string)obj;
return _id.CompareTo(obj1);
}
throw new ArgumentException();
}
public override bool Equals(object obj)
{
if (obj == null)
return false;
if (obj is string)
{
return (string)obj == _id;
}
if (obj is TextFileContentPartIdentity)
{
var obj1 = obj as TextFileContentPartIdentity;
return _id == obj1._id;
}
//comparing 2 ContentPartIdentityBase objs will end up
here with obj typed as ContentPartIdentityBase.
//it will fall through to here and ToString() is the only
valid way to compare them
if (obj is ContentPartIdentityBase)
{
bool test = obj.ToString() == ToString();
return test;
}
throw new ArgumentException();
}
public override int GetHashCode()
{
return _id.GetHashCode();
}
```

```
public void GetObjectData(SerializationInfo info,
StreamingContext context)
{
info.AddValue(ID_INFO, _id);
}
public override string ToString()
{
return_id;
}
#endregion
#region public operators...
public static explicit operator
TextFileContentPartIdentity(string a)
{
return new TextFileContentPartIdentity(a);
}
public static explicit operator
string(TextFileContentPartIdentity a)
{
return a._id;
}
public static bool operator ==(TextFileContentPartIdentity
a, TextFileContentPartIdentity b)
{
if (ReferenceEquals(null, a)
return ReferenceEquals(null, b);
return a._id == b._id;
}
public static bool operator !=(TextFileContentPartIdentity
a, TextFileContentPartIdentity b)
{
return a._id != b._id;
}
public static bool operator ==(TextFileContentPartIdentity
a, string b)
{
return a. Equals(b);
}
public static bool operator !=(TextFileContentPartIdentity
a, string b)
{
return !a.Equals(b);
}
#endregion
}
```

Tarefa 7: Habilitar a geração de identidades de parte de conteúdo

O conector do gateway de dados precisa gerar um novo identificador de registro externo ao criar conteúdo no sistema. O método `GenerateContentPartIdentity()` desempenha esta função. Faça o seguinte para permitir que o conector gere identidades de parte de conteúdo.

1. Abra a classe SqlDatasource e localize o método `GenerateContentPartIdentity()`.

2. Add the following code:

```
public ContentPartIdentityBase
GenerateContentPartIdentity()
{
var identifier = Guid.NewGuid();
using (var conn = new SqlConnection(connectionString))
using (var cmd = conn.CreateCommand())
{
conn.Open();
//INSERT new record that will be updated with all the
fields in the Set method.
cmd.CommandText = $"INSERT INTO Records (ID) VALUES
('{identifier}');";
cmd.ExecuteNonQuery();
}
return new ContentPartIdentity(identifier.ToString());
}
```

   Este código permite que o método gere um novo GUID e use seu valor como o identificador de uma nova instância de SqlContentPartIdentity. O conteúdo externo típico armazena a lógica de execução para gerar novos registros no sistema externo, e não dentro da Datasource propriamente dita. Quando você adiciona um novo registro ao aplicativo de gateway de dados no Archer, o método insere um novo identificador na tabela de registros que representa o novo registro.

Tarefa 8: Habilitar a análise de identidades de parte de conteúdo

O conector do gateway de dados precisa analisar uma representação de string de um identificador de registro externo em uma instância de ContentPartIdentityBase para que o sistema do gateway de dados consiga identificar o registro. O método `ParseContentIdentity()` desempenha

esta função. Faça o seguinte para permitir que o conector analise identidades de parte de conteúdo.

1. Localize o método `ParseContentPartIdentity(string)`.
2. Atualize o método com o seguinte código:
   ```
   public ContentPartIdentityBase
   ParseContentPartIdentity(string contentPartIdentity)
   {
   return new ContentPartIdentity(contentPartIdentity);
   }
   ```
   Este código permite que o método crie uma nova instância da classe ContentPartIdentity e passe-a para o identificador de string.

Tarefa 9: Permitir a clonagem de Datasource

Para garantir a segurança do thread ao processar conteúdo externo, o sistema do gateway de dados clona os conectores de Datasource quando o sistema solicita novas instâncias. A clonagem de Datasource diminui o tempo necessário para criar uma nova instância de conector. O método `Datasource Clone()` desempenha esta função. Faça o seguinte para permitir que o conector prepare um clone da instância atual e de suas propriedades.

1. Localize o método `Clone()`.
2. Atualize o método com o seguinte código:
   ```
   public IContentDatasource Clone()
   {
   SqlDatasource newObject = MemberwiseClone() as
   SqlDatasource; returnnewObject;
   }
   ```

Lendo dados

O conector do gateway de dados usa uma implementação da interface IContentDataReader para ler dados. O leitor de dados lê dados do armazenamento de conteúdo e pode receber informações de ID e de campo após a criação para pré-carregar registros para o processamento eficiente da leitura.

Realize as tarefas a seguir para permitir que seu conector leia dados.

Tarefa 10: Criar DTO (Data Transfer Object, objeto de transferência de dados) do registro

Uma classe DTO contendo propriedades de registro representa cada registro ao ler e gravar dados. Faça o seguinte para criar a classe DTO do registro:

1. Adicione ao projeto uma nova classe chamada SqlRecord.
2. Implemente a classe com o seguinte código:

```
public class SqlRecord
{
public SqlRecord(ContentPartIdentity identity)
{
Identity = identity;
}
public ContentPartIdentity Identity { get; }
public string Name { get; set; }
public string Description { get; set; }
public int Count { get; set; }
}
```

Tarefa 11: Criar o leitor de dados

O leitor de dados retorna o SqlContentPart apropriado quando lido. A estrutura do gateway de dados inclui a classe ContentDataReader por padrão, que atende aos requisitos básicos do leitor de dados e tem ganchos para criar identidades de parte de conteúdo e ler campos. Caso seja necessário criar um leitor de dados personalizado, ignore esta tarefa e realize o procedimento descrito na Tarefa 12. Caso contrário, faça o seguinte para criar o leitor de dados usando a classe ContentDataReader.

1. Abra a classe SqlDatasource.
2. Adicione o seguinte código para fornecer lógica para a leitura dos campos durante a criação de uma instância da classe ContentDataReader, em que SQL é uma string que representa o nome da conexão ou um alias do conector:

```
var reader = new ContentDataReader <SqlRecord>("SQL",
records, fieldIds, x => new ContentPartIdentity(x.key));
```

3. Adicione código semelhante ao exemplo a seguir para configurar a leitura para cada campo, em que Name é o nome da coluna no banco de dados externo que representa o campo Name no aplicativo do gateway de dados:

```
reader.HandleField("name", x => newTextFieldValue { Data =
x.Name });
```

Tarefa 12 (opcional): Criar um leitor de dados de conteúdo personalizado

Um leitor de dados de conteúdo personalizado itera registros, cria campos e monta uma resposta. Faça o seguinte para implementar um leitor de dados personalizado:

1. Implemente a interface ArcherTech.Datasource.Common.IContentDataReader.

2. Adicione o seguinte código:

```
private readonly IEnumerator<TextFileContentPart> _parts;
public TextFileDataReader(IEnumerable<TextFileRecord>
records, IEnumerable<string> fieldIds)
{
var parts = new List<TextFileContentPart>();
foreach (TextFileRecordrecord in records)
{
ContentPartIdentityBase identity = new
TextFileContentPartIdentity(record.Identity);
var part = new TextFileContentPart(identity);
parts.Add(part);
foreach(string fieldid in fieldIds.Distinct())
{
ContentPartDataField dataField;
switch (fieldid.ToLower())
{
case "name":
dataField = new ContentPartDataField<TextFieldValue>
{
FieldValue = new TextFieldValue { Data = record.Name }
};
break;
case "description":
dataField = new ContentPartDataField<TextFieldValue>
{
FieldValue = new TextFieldValue { Data =
record.Description }
};
break;
case "count":
dataField = new ContentPartDataField<NumericFieldValue>
{
FieldValue = new NumericFieldValue { Data = record.Count }
};
```

```
break;
default:
throw new ArgumentException($"No field handler found for
{fieldid.ToLower()}");
}
dataField.Alias = "TF";
dataField.Identity = identity;
dataField.SourceFieldId = fieldid;
part.ContentParts.Add(dataField);
}
}
_parts = parts.GetEnumerator();
}
```

A classe Datasource gera uma lista de registros de arquivos de texto ao criar a instância do leitor de dados. Este código adiciona um construtor à classe, que pega um enumerável de TextFileRecords e um enumerável de IDs de campo e analisa TextFileRecords em um DTO de TextFileContentPart, o que ajuda a obter os dados de campo específicos quando os registros são lidos. O código também define um enumerador local para ajudar na iteração dos resultados.

3. Adicione o seguinte código para verificar se partes adicionais devem ser lidas:

```
public bool Read()
{
return _parts.MoveNext();
}
```

4. Adicione o seguinte código:

```
public IEnumerable<ContentPartDataField<T>>
ReadData<T>(IEnumerable<string> fieldIds) where T :
FieldValueBase
{
var dataFields = new List<ContentPartDataField<T>>();
if (_parts.Current != null)
dataFields.AddRange(_parts.Current.ContentParts.OfType<ContentPartDat
return dataFields;
}
```

Ao ler os dados, o método `ReadData()` fornece uma lista de IDs de campo a serem retornados com o registro. O valor genérico <T> do método identifica o tipo de campo de dados a ser retornado. Este código recupera as partes de conteúdo do mesmo tipo do registro atual.

5. Adicione o seguinte código para notificar o leitor de dados para fechar quaisquer conexões:

```
public void Close() {}
```

6. Adicione o seguinte código para implementar a propriedade CurrentIdentity e retornar a identidade atual no enumerador de partes de conteúdo:

```
public ContentPartIdentityBase CurrentIdentity =>
_parts.Current?.Identity;
```

Tarefa 13: Preparar e retornar o leitor de dados

Para obter uma lista de tipos de campo com suporte por conectores SQL flexível e conectores personalizados, consulte Configurar conexões de gateway de dados (API RESTful).

Quando o sistema do gateway de dados solicita um leitor de dados, o conector deve ler os registros do banco de dados para a lista de identificadores passada para o método `Get()`. Faça o seguinte para permitir que o conector prepare e retorne o leitor de dados.

1. Localize o método `Get()` na classe SqlDatasource e adicione o código a seguir, em que SQL é uma string que representa o nome da conexão ou um alias do conector:

```
public IContentDataReader
Get(IEnumerable<ContentPartIdentityBase> contentIds,
IList<string> fieldIds)
{
List<SqlRecord> records = new List<SqlRecord>();
using (SqlConnection conn = new
SqlConnection(connectionString))
using (SqlCommand cmd = conn.CreateCommand())
{
conn.Open();
cmd.CommandText = $"SELECT id, {string.Join(",",
fieldIds)} from Records WHERE id IN
({string.Join(",",contentIds.Select(x => $"'{x}'"))});";
using (var dr = cmd.ExecuteReader())
{
while (dr.Read())
{
SqlRecord r = new SqlRecord(new
ContentPartIdentity(dr.GetString(0)));
if (fieldIds.Contains("name"))
r.Name = dr.GetString(dr.GetOrdinal("name"));
if(fieldIds.Contains("description"))
r.Description =
dr.GetString(dr.GetOrdinal("description"));
if(fieldIds.Contains("count"))
```

```
r.Count = dr.GetInt32(dr.GetOrdinal("count"));
records.Add(r);}
}
}
var dataReader =
new ContentDataReader<SqlRecord>("SQL", records, fieldIds,
x => x.Identity);
dataReader.HandleField("name", x => new TextFieldValue
{ Data= x.Name });
dataReader.HandleField("description", x => new
TextFieldValue { Data = x.Description });
dataReader.HandleField("count", x => new NumericFieldValue
{ Data = x.Count });
return dataReader;
}
```

Esse código usa um enumerável de instâncias de ContentPartIdentityBase e uma lista de strings que representam os IDs de campo a serem retornados para cada registro. Em seguida, o método lê os registros SQL usando a leitura da string de conexão com base no modelo de configuração descrito na Tarefa 3, e transfere os registros e os IDs de campo para uma nova instância de ContentDataReader, que é retornada ao chamador.

2. Localize o método `GetAsync()` e adicione o seguinte código:

```
public Task<IContentDataReader>
GetAsync(IEnumerable<ContentPartIdentityBase> contentIds,
IList<string> fieldIds)
{
return Task.Run(() => Get(contentIds, fieldIds));
}
```

Este código desempenha a mesma função que o método `Get()` na Etapa 1, mas é tratado como tarefa assíncrona.

Gravando dados

O conector do gateway de dados fornece ao sistema do gateway de dados uma implementação dedicada de IContentDataWriter para gravar dados no armazenamento de conteúdo externo. Realize as tarefas a seguir para permitir que seu conector grave dados.

Tarefa 14: Criar o gravador de dados

A estrutura do gateway de dados inclui a classe ContentDataWriter por padrão, que atende aos requisitos básicos de gravador de dados e tem ganchos para gravar campos do Archer no DTO e publicar o DTO na fonte

de dados externa. Caso seja necessário criar um gravador de dados personalizado, ignore esta tarefa e realize o procedimento descrito na Tarefa 15. Caso contrário, faça o seguinte para criar o gravador de dados usando a classe ContentDataWrriter.

1. Abra a classe SqlDatasource.

2. Adicione o seguinte código para fornecer uma lógica para a leitura dos campos de Archer ao criar uma instância da classe ContentDataReader:

```
var writer = new ContentDataWriter<SqlRecord>(x =>
new SqlRecord((ContentPartIdentity)x),
x => SaveRecord(x));
```

3. Adicione código semelhante ao seguinte exemplo para configurar a gravação de cada campo no DTO:

```
writer.HandleField<TextFieldValue>("name", (field, record)
=> record.Name = field.Data);
```

Observação: A classe também fornece um gancho para confirmar a gravação depois de concluir todos os registros, se compatível com a fonte de dados externa.

Tarefa 15 (opcional): Criar um gravador de dados de conteúdo personalizado

Um gravador de dados personalizado segue uma sequência específica de eventos para permitir que o conector grave registros individuais e confirme todas as alterações no final do processamento. Faça o seguinte para implementar um gravador de dados de conteúdo personalizado.

1. Implemente a interface IContentDataWriter.

2. Adicione o seguinte código:

```
private readonly ContentWriteResultCollection _results;
private readonly IList<TextFileRecord> _records;
private TextFileRecord _currentRecord;
private ContentWriteResult _currentResult;
private readonly string _filePath;
public TextFileDataWriter(string filePath)
{
_filePath = filePath;
_results = new ContentWriteResultCollection();
_records = newList<TextFileRecord>();
}
```

Este código cria um construtor que pega o caminho de arquivo que a fonte de dados principal leu da configuração e configura 2 coletas. A primeira coleta todos os resultados da gravação dos registros de salvar, e a outra cria um

conjunto de instâncias de TextFileRecord a serem mantidas no arquivo de texto. O código também adiciona variáveis de membros privados para manter o registro atual que está sendo gravado, os resultados de gravação atuais que estão sendo criados e o caminho de arquivo para o arquivo de texto.

3. Adicione o seguinte código:

```
public void StartRecord(ContentSaveContext context)
{
_currentRecord = new TextFileRecord(context.Identity as
TextFileContentPartIdentity);
_currentResult = new ContentWriteResult
{
Identity = context.Identity,
Exceptions = new List<Exception>(),
IsSuccessful = true
};
}
```

Este código adiciona o método `StartRecord()` que é chamado primeiro na sequência e passa um objeto de contexto com informações suficientes para o gravador de dados preparar a gravação de dados de campo. Esta é uma oportunidade de criar uma nova instância de TextFileRecord para coletar os dados de campo do registro atual e criar um objeto de resultado de gravação para rastrear o sucesso e as exceções de gravações do registro atual.

4. Adicione o seguinte código:

```
public ContentWriteResult
ProcessFields<T>(IEnumerable<ContentPartDataField<T>>
dataFields)
where T : FieldValueBase
{
var exceptions = new List<Exception>();
// Each ContentPartDataField represents a type of field
and its value.
// The data field's source field id is the field
identifier from the external system.
// Parse the value from each data field based on its
source field id and update.
// the current TextFileRecord.
foreach (ContentPartDataField<T> dataField in dataFields)
{
switch (dataField.SourceFieldId.ToLower())
{
case "name":
var nameDataField = dataField as
```

```
ContentPartDataField<TextFieldValue>;
if (nameDataField == null)
{
exceptions.Add(new ArgumentException($"Name field is not
of type
{typeof(TextFieldValue).Name}."));
}
_currentRecord.Name = nameDataField.FieldValue.Data;
break;
case "description":
var descriptionDataField = dataField
asContentPartDataField<TextFieldValue>;
if (descriptionDataField == null)
{
exceptions.Add(new ArgumentException($"Description field
is not of type
{typeof(TextFieldValue).Name}."));
}
_currentRecord.Description =
descriptionDataField.FieldValue.Data;
break;
case "count":
var orderDataField = dataField as
ContentPartDataField<NumericFieldValue>;
if (orderDataField == null)
{
exceptions.Add(new ArgumentException($"Count field is not
of type
{typeof(TextFieldValue).Name}."));
}
_currentRecord.Count = (int)
orderDataField.FieldValue.Data.Value;
break;
}
}
// Add the exceptions for the data field type to the
result for the entire record.
_currentResult.Exceptions.AddRange(exceptions);
_currentResult.IsSuccessful = !
_currentResult.Exceptions.Any();
// Return the write result for just this collection of
fields
return new ContentWriteResult
{
```

```
Exceptions = exceptions,
Identity = _currentResult.Identity,
IsSuccessful = !exceptions.Any()
};
}
```

Esse código adiciona o método `ProcessFields()`, que recebe um conjunto de objetos ContentPartDataField<T> da estrutura do gateway de dados e é chamado várias vezes para cada tipo de campo (texto, numérico, data etc.) até que todos os tipos de campo do registro atual tenham sido gravados.

5. Adicione o seguinte código:

```
public void EndRecord()
{
_records.Add(_currentRecord);
_results.Add(_currentResult);
}
```

Esse código adiciona o método `EndRecord()`, que permite que o gravador de dados grave um único registro no sistema, se necessário, ou realize uma limpeza entre os registros. Para TextFileDataWriter, o método adiciona o TextFileRecord atual à conjunto de registros local e o resultado da gravação atual ao conjunto local.

6. Adicione o seguinte código:

```
public ContentWriteResultCollection Commit()
{
// Serialize the text file records to JSON and save to
file.
MemoryStream ms = newMemoryStream();
DataContractJsonSerializer ser = new
DataContractJsonSerializer(typeof(IEnumerable<TextFileRecord>));
ser.WriteObject(ms, _records);
byte[] json = ms.ToArray();
ms.Close();
File.WriteAllBytes(_filePath, json);
// Return the write results.
return_results;
}
```

Esse código adiciona o método `Commit()`, que serializa os TextFileRecords criados durante as etapas anteriores para JSON, grava todos no arquivo e retorna os resultados da gravação para o sistema.

7. Adicione o seguinte código:

```
public Task<ContentWriteResultCollection> CommitAsync()
```

```
{
return Task.Run(() => Commit());
}
```
Este código adiciona o método `CommitAsync()`, que desempenha a mesma função que `Commit()`, mas é tratado como tarefa assíncrona.

Tarefa 16: Preparar e retornar o gravador de dados

O sistema do gateway de dados solicita o gravador de dados do conector Datasource. Faça o seguinte para permitir que o conector crie e retorne o gravador de dados.

1. Abra a classe SqlDatasource.
2. Localize o método `GetWriter()` e adicione o seguinte código:
```
public IContentDataWriter GetWriter()
{
var writer = new ContentDataWriter<SqlRecord>(x =>
new SqlRecord((ContentPartIdentity)x),
WriteSqlRecord);
writer.HandleField<TextFieldValue>("name", (field, record)
=> record.Name = field.Data);
writer.HandleField<TextFieldValue>("description", (field,
record) => record.Description = field.Data);
writer.HandleField<NumericFieldValue>("count", (field,
record) => record.Count =
Convert.ToInt32(field.Data.GetValueOrDefault()));
returnwriter;
}
```
Este código retorna uma nova ContentDataWriter com a string de conexão apropriada.
3. Adicione o seguinte código:
```
private void WriteSqlRecord(SqlRecord record)
{
using (var conn = new SqlConnection(connectionString))
using (var cmd = conn.CreateCommand())
{
conn.Open();
cmd.CommandText = //ALWAYS UPDATE BECAUSE NEW RECORDS ARE
INSERTED IN GenerateContentPartIdentity()
$"UPDATE Records SET name='{record.Name}',
description='{record.Description}',
count={record.Count} where
id='{record.Identity.PartId}';";
```

```
cmd.ExecuteNonQuery();
}
}
```

Este código grava o registro no banco de dados.

Tarefa 17: Implementar a exclusão de conteúdo

O conector do gateway de dados pode excluir dados de registros do banco de dados por lote. Faça o seguinte para implementar a exclusão de conteúdo.

1. Abra a classe SqlDatasource.
2. Localize o método `Delete()` e adicione o seguinte código:

```
public void Delete(IEnumerable<ContentPartIdentityBase>
identities)
{
using (var conn = new SqlConnection(connectionString))
using (var cmd = conn.CreateCommand())
{
conn.Open();
cmd.CommandText = $"DELETE FROM Records WHERE id IN
({string.Join(",",
identities.Select(x => $"'{x}'"))})";
cmd.ExecuteNonQuery();
}
}
```

Pesquisando conteúdo

A API de pesquisa do gateway de dados exige que o conector forneça uma implementação de IContentSearchFilter. Esse filtro aceita SearchExpressionTree e analisa a expressão para criar uma consulta e, depois, executá-la na origem externa. A resposta da consulta fornece as identidades de partes de conteúdo dos registros correspondentes, que a API carrega ao processar os resultados da pesquisa. Como a análise de SearchExpressionTree pode ser complexa, a estrutura do gateway de dados fornece a classe SearchQueryBuilder para ajudar a facilitar o processo de consultas baseadas em texto, inclusive strings de consulta SQL ou HTTP. Em alguns casos, pode ser necessária uma análise personalizada de SearchExpressionTree. Realize as tarefas a seguir para permitir que seu conector pesquise conteúdo.

Tarefa 18: Construir SearchQueryBuilder

SearchQueryBuilder é uma série de objetos IConditionBuilder que definem cada parâmetro de pesquisa específico. SearchQueryBuilder inclui implementações de IConditionBuilder para cenários comuns, como "texto contém" e "numérico é igual". Você pode encontrar as implementações de IConditionBuilder no namespace 'ArcherTech.Datasource.Common.Domain.Search.Conditions'.

Você pode estender SearchQueryBuilder e as classes IConditionBuilder fornecidas por meio de sobreposições de método ou injeção de comportamento para criar uma implementação de IConditionBuilder personalizada.

Faça o seguinte para construir SearchQueryBuilder para sua implementação.

1. Abra a classe SearchQueryBuilder.
2. Passe os construtores de condição que compõem a consulta desejada. Substitua ou complemente os construtores de condição no exemplo a seguir, conforme necessário:
```
var queryBuilder = new SearchQueryBuilder(
new NumericEqualConditionBuilder(),
new NumericNotEqualConditionBuilder(),
new NumericGreaterThanConditionBuilder(),
new NumericLessThanConditionBuilder(),
new TextContainsConditionBuilder(), // new
SqlLikeConditionBuilder(),
new TextNotContainsConditionBuilder(), // new
SqlNotLikeConditionBuilder(),
new TextEqualConditionBuilder(),
new TextExactConditionBuilder(),
new TextNotEqualConditionBuilder(),
new TextNotExactConditionBuilder())
```
3. (Opcional) Especifique um construtor de condição para um único campo, se necessário:
```
queryBuilder.SetFieldConditionBuilder("name", new
NameConditionBuilder());
```

Tarefa 19: Implementar IContentSearchFilter

A classe IContentSearchFilter analisa uma árvore de expressão usando SearchQueryBuilder ou lógica personalizada e, em seguida, executa essa consulta na fonte de dados externa. A classe retorna os resultados da pesquisa dentro de um objeto SearchFilterResponse, que contém as

identidades de parte de conteúdo dos registros correspondentes. Faça o seguinte para implementar IContentSearchFilter.

1. Abra a classe SqlDatasource.
2. Localize o método GetFilter() e adicione o seguinte código:

```
var queryBuilder = new SearchQueryBuilder(
new NumericEqualConditionBuilder(),
new NumericNotEqualConditionBuilder(),
new NumericGreaterThanConditionBuilder(),
new NumericLessThanConditionBuilder(),
new SqlLikeConditionBuilder(),
new SqlNotLikeConditionBuilder(),
new TextEqualConditionBuilder(),
new TextExactConditionBuilder(),
new TextNotEqualConditionBuilder(),
new TextNotExactConditionBuilder());
return new SqlContentSearchFilter(connectionString,
queryBuilder);
```

   Este código retorna o objeto IContentSearchFilter apropriado.
3. Adicione o seguinte código:

```
public class SqlContentSearchFilter : IContentSearchFilter
{
private readonly string connectionString;
private readonly SearchQueryBuilder queryBuilder;
public SqlContentSearchFilter(string connectionString,
SearchQueryBuilder queryBuilder)
{
this.connectionString = connectionString;
this.queryBuilder = queryBuilder;
}
public SearchFilterResponse
EvaluateNode(SearchExpressionTree searchExpression)
{
List<ContentPartIdentityBase> identities = new
List<ContentPartIdentityBase>();
StringBuilder sql = new StringBuilder();
sql.Append("SELECT id FROM Records WHERE
").Append(queryBuilder.BuildQuery(searchExpression));
using (var conn = new SqlConnection(connectionString))
using (var cmd = conn.CreateCommand())
{
conn.Open();
cmd.CommandText = sql.ToString();
using (var dr = cmd.ExecuteReader())
{
```

```
while (dr.Read())
{
identities.Add(new ContentPartIdentity(dr.GetString(0)));
}
}
}
return new SearchFilterResponse { ContentPartIdentityBases
= identities };
}
}
```

Este código implementa o IContentSearchFilter.

Tarefa 20: (opcional): Implementar conexão de teste

Os usuários podem testar suas conexões de gateway de dados quando a configuração do conector inclui a funcionalidade Testar conexão. Para implementar essa funcionalidade do gateway de dados, inclua o seguinte no conector personalizado:

1. Abra a classe SqlDatasource.
2. Atualize a classe para implementar a interface ArcherTech.Datasource.Common.Interfaces.ITestableDatasource.
3. Implemente o método TestConnection() de acordo com a origem externa e retorne TestConnectionResult com o resultado.

Tarefa 21: Adicionar uma conexão para o conector do gateway de dados

Use a API do gateway de dados para adicionar uma conexão ao conector dele.

A conexão deve incluir:

- O nome completo do tipo de Datasourceclass: (ArcherTech.Datasource.SqlDemo.SqlDatasource).
- Nome exclusivo (alias) de Datasource. (Isso vincula as fontes de dados aos campos em Archer.)
- As propriedades personalizadas (pares de chave-valor) a serem transferidas para o conector.

Para obter mais informações, consulte Adicionar conexão.

Tarefa 22: Mapear campos

Use a API do gateway de dados para mapear os campos de registro externo para definições de campo Archer.

Crie um aplicativo com 3 campos com os seguintes atributos em Archer e altere o mapa de campos referente a cada campo de modo a apontar para um campo externo:

A tabela a seguir descreve os atributos.

Nome	**Tipo**
Nome	Texto
Descrição	Texto
Contagem	Numérico

Para obter mais informações, consulte Adicionar ou editar mapas de campo.

Tarefa 23: Implementar o conector

Antes de implementar o conector, você deve construir o projeto .NET usando uma plataforma de solução ativa x64. A construção do projeto cria um arquivo ArcherTech.Datasource.SqlDemo.dll na pasta de saída do projeto. Faça o seguinte para implementar o conector:

1. Copie este arquivo para 3 lugares na instalação do aplicativo da Web Archer. Se você precisar que o conector funcione com o mecanismo de trabalho, deverá copiar o arquivo para os mesmos locais de diretório nos servidores do mecanismo de trabalho.
    - [diretório raiz da Web]\bin\DatasourcePlugins\
    - [diretório raiz da Web]\Api\bin\DatasourcePlugins\
    - [diretório raiz da Web]\ContentApi\bin\DatasourcePlugins\
    - [diretório raiz da Web]\MobileAPI\bin\DatasourcePlugins\
2. Se Archer estiver em execução, reinicie o IIS.
3. Para testar o conector, navegue até o aplicativo com os campos mapeados na Tarefa 21 e faça o seguinte:
    - Adicione um novo registro Você deve ver o registro agora no banco de dados.

- Atualize o registro e observe como os valores no banco de dados mudam.
- Faça uma pesquisa para localizar qualquer registro que contenha texto no campo de nome.
- Exclua o registro para confirmar que ele foi removido do banco de dados.