Endeca® Platform Services

Forge API Guide for Perl

Version 6.0.3 • June 2012

Contents

Copyright and disclaimer

Preface

Oracle Endeca's Web commerce solution enables your company to deliver a personalized, consistent customer buying experience across all channels — online, in-store, mobile, or social. Whenever and wherever customers engage with your business, the Oracle Endeca Web commerce solution delivers, analyzes, and targets just the right content to just the right customer to encourage clicks and drive business results.

Oracle Endeca Commerce is the most effective way for your customers to dynamically explore your storefront and find relevant and desired items quickly. An industry-leading faceted search and Guided Navigation solution, Oracle Endeca Commerce enables businesses to help guide and influence customers in each step of their search experience. At the core of Oracle Endeca Commerce is the MDEX Engine,™ a hybrid search-analytical database specifically designed for high-performance exploration and discovery. The Endeca Content Acquisition System provides a set of extensible mechanisms to bring both structured data and unstructured content into the MDEX Engine from a variety of source systems. Endeca Assembler dynamically assembles content from any resource and seamlessly combines it with results from the MDEX Engine.

Oracle Endeca Experience Manager is a single, flexible solution that enables you to create, deliver, and manage content-rich, cross-channel customer experiences. It also enables non-technical business users to deliver targeted, user-centric online experiences in a scalable way — creating always-relevant customer interactions that increase conversion rates and accelerate cross-channel sales. Non-technical users can control how, where, when, and what type of content is presented in response to any search, category selection, or facet refinement.

These components — along with additional modules for SEO, Social, and Mobile channel support — make up the core of Oracle Endeca Experience Manager, a customer experience management platform focused on delivering the most relevant, targeted, and optimized experience for every customer, at every step, across all customer touch points.

About this guide

This reference describes the classes and methods you can incorporate in a Perl manipulator component. A Perl manipulator component uses Perl to efficiently manipulate source records as part of Forge's data processing. For example, pipeline developers can use a Perl manipulator to add, remove, and reformat properties, join record sources, and so on.

Who should use this guide

This reference is intended for developers who are building Forge pipelines using Endeca Developer Studio.

Conventions used in this guide

This guide uses the following typographical conventions:

Code examples, inline references to code elements, file names, and user input are set in `monospace` font. In the case of long lines of code, or when inline monospace text occurs at the end of a line, the following symbol is used to show that the content continues on to the next line: ¬

When copying and pasting such examples, ensure that any occurrences of the symbol and the corresponding line break are deleted and any remaining space is closed up.

Contacting Oracle Support

Oracle Support provides registered users with important information regarding Oracle Endeca software, implementation questions, product and solution help, as well as overall news and updates.

You can contact Oracle Support through Oracle's Support portal, My Oracle Support at *https://support.oracle.com*.

Chapter 1

Introduction to the Forge API

This guide describes the classes and methods you can incorporate in a Perl manipulator component.

About the Perl manipulator

A Perl manipulator component uses Perl to efficiently manipulate source records as part of Forge's data processing. For example, pipeline developers can use a Perl manipulator to add, remove, and reformat properties, join record sources, and so on.

The Forge API is the interface between a Perl manipulator and the Forge Execution Framework . All pipeline components (record adapter, spider, indexer adapter, and so on) plug into the Forge Execution Framework using one of two methods: `next_record` or `get_records`. For all components except the Perl manipulator, these two methods are hidden from the pipeline developer.

Before a component such as an indexer adapter processes records, it gets them from an upstream component by calling `next_record` or `get_records` via the Forge Execution Framework. See the topic on understanding Forge and downstream record processing for an explanation of this process.

However, for the Perl manipulator, the `next_record` and `get_records` methods are exposed to you as methods that you can override. These methods allow the Perl manipulator to plug into the Forge Execution Framework and make calls to upstream pipeline components during record processing.

There are the four overrideable methods available to a Perl manipulator. They are overrideable because the Perl manipulator component in Developer Studio provides an empty implementation of each one. If you run the default implementation, the methods do nothing. It is your custom implementation that overrides the default and defines how a Perl manipulator behaves.

Each method can be called at different stages of record processing. At a minimum, one method (either `get_records` or `next_record`) is required to retrieve records.

- `EDF::Manipulator::prepare`—called before record processing starts to perform setup and initialization tasks. Optional.
- `EDF::Manipulator::next_record`—called during record processing when a downstream pipeline component requests the next record from the Perl manipulator. This method accesses any record sources specified on the Sources tab of your Perl manipulator. More commonly used than `get_records`.
- `EDF::Manipulator::get_records`—called during record processing when a downstream pipeline component requests the set of records matching a given key. This method accesses any record sources specified on the Sources tab of your Perl manipulator.
- `EDF::Manipulator::finish`—called when record processing is complete. Typically performs clean up or logging tasks. Optional.

Within these four overrideable methods, you can use classes and methods in the EDF namespace that Endeca provides to perform record manipulation. In other words, you implement the four overrideable methods of `EDF::Manipulator` using methods and classes in the EDF namespace such as `EDF::Record`, `EDF::PVal`, `EDF::DVal`, and so on. See the classes available to a Perl manipulator for an overview of each class.

Creating a Perl manipulator

See the *Developer Studio Help* for information about creating, modifying, or removing a Perl manipulator component from your pipeline.

Classes available to a Perl manipulator

If Forge encounters a Perl manipulator component while processing your pipeline, it creates an instance of the `EDF::Manipulator` class.

The `EDF::Manipulator` works with `EDF::RecordSource` to retrieve records using the classes `EDF::RecordKey` and `EDF::KeyColumn` to identify records. The `EDF::Manipulator` manipulates the structure of records with the `EDF::Record`, `EDF::PVal`, and `EDF::DVal` classes. In addition, there are also several static methods available in the EDF name space for logging.

Here is a summary of the available classes:

Class	Description
`EDF::DVal`	Represents a dimension value tagged to a record or contained in an `EDF::RecordKey`.
`EDF::KeyColumn`	Represents a value for comparison within an `EDF::RecordKey`.
`EDF::Manipulator`	Represents a Perl manipulator component in Forge.
`EDF::PVal`	Represents a property value on a record or in an `EDF::RecordKey`.
`EDF::Record`	Describes a record.
`EDF::RecordKey`	Contains the information necessary to identify records for record selection and joins.
`EDF::RecordSource`	Represents a record source specified on the Sources tab of the Perl Manipulator editor.
Static methods	Contains methods for logging different levels of messages (ERR, INF, WRN).

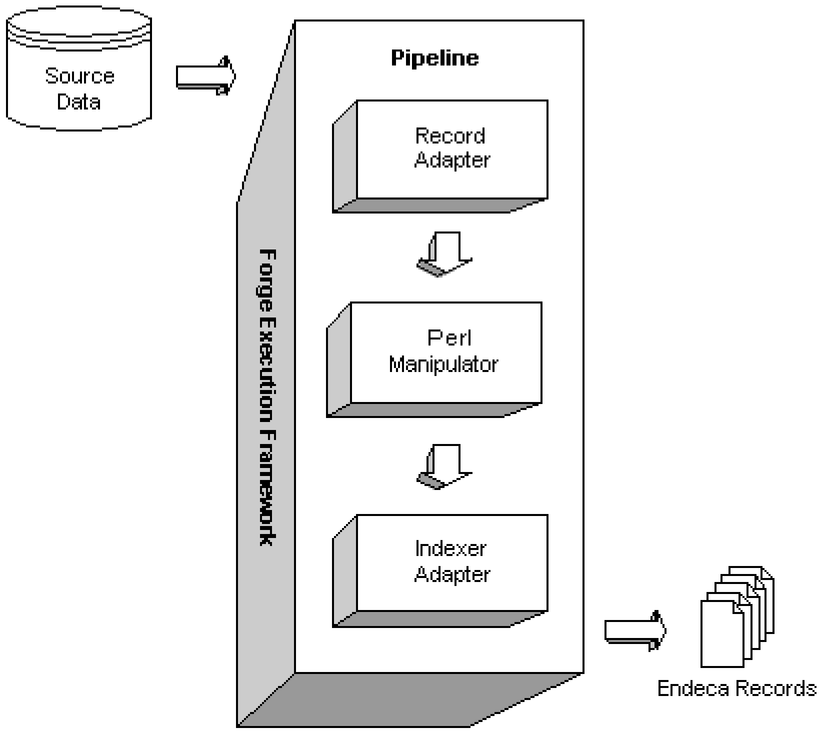
Understanding Forge and downstream record processing

The pipeline metaphor suggests all data moves downstream through a pipeline during processing. It is important to understand that although the term pipeline suggests that record processing occurs in a downstream order (a push scenario beginning with source data and ending with indexed Endeca records), Forge actually processes records by requesting records from upstream components (a pull scenario) to retrieve records as necessary.

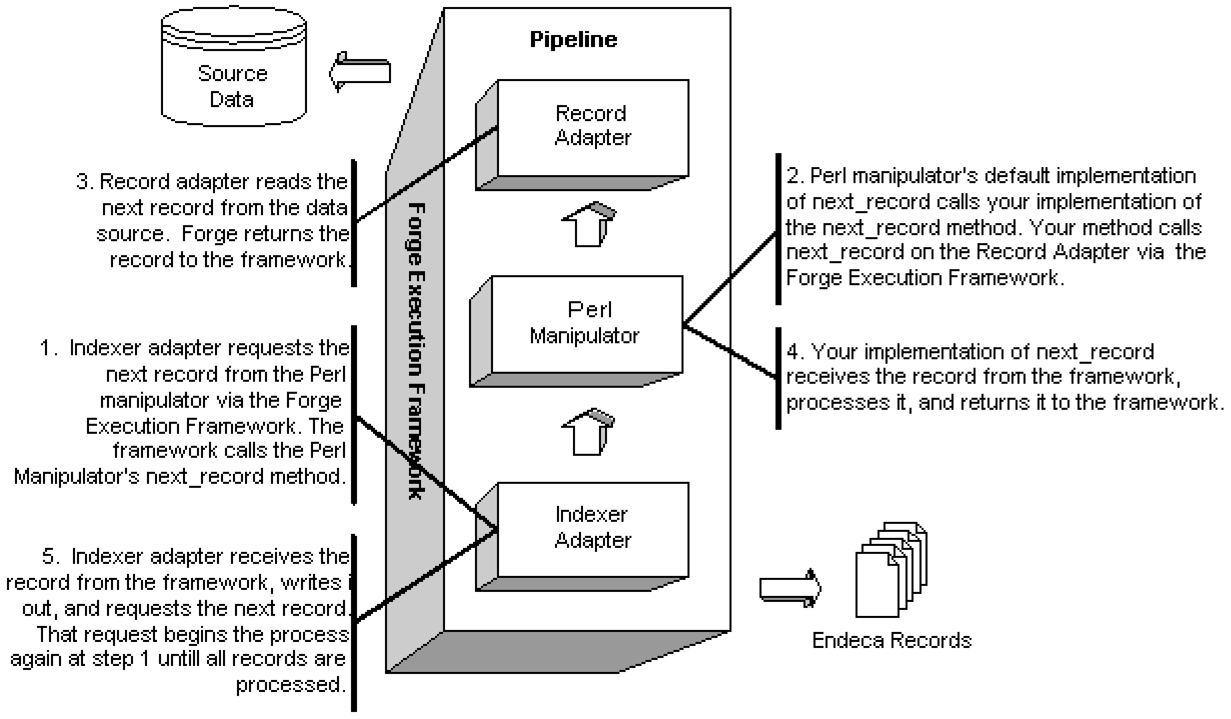
Pipeline components, such as a record adapter, Perl manipulator, indexer adapter, spider, and so on, call backwards up a pipeline, either requesting a record one at a time using the `next_record` method, or requesting all records that match a key using the `get_records` method. Forge then returns the records downstream to the requesting component for processing.

When you write the `EDF::Manipulator::next_record` or `EDF::Manipulator::get_records` method for a Perl manipulator, you are defining how the Forge Execution Framework retrieves the records from the Perl manipulator and how the framework returns them to the downstream component.

It's useful to contrast downstream record flow with upstream method calls in the diagrams below. The first diagram shows the conceptual explanation of downstream processing. Records flow from a source database through the pipeline, and Forge produces Endeca records as a result.



The second diagram shows each component in the pipeline calling `next_record` through the Forge Execution Framework to make upstream requests for records. The upstream requests are represented in steps 1, 2, and 3. The Forge Execution Framework returns the records to the requesting component. The downstream record flow is represented in steps 4 and 5.


Source Data
Pipeline
Record Adapter
Perl Manipulator
Indexer Adapter
Forge Execution Framework
Endeca Records
3. Record adapter reads the next record from the data source. Forge returns the record to the framework.
1. Indexer adapter requests the next record from the Perl manipulator via the Forge Execution Framework. The framework calls the Perl Manipulator's next_record method.
5. Indexer adapter receives the record from the framework, writes it out, and requests the next record. That request begins the process again at step 1 untill all records are processed.
2. Perl manipulator's default implementation of next_record calls your implementation of the next_record method. Your method calls next_record on the Record Adapter via the Forge Execution Framework.
4. Your implementation of next_record receives the record from the framework, processes it, and returns it to the framework.

About the Forge Execution Framework

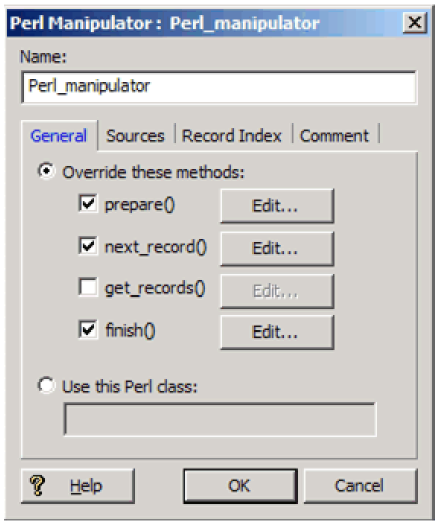
The Forge Execution Framework is the layer of Forge that runs a Forge pipeline.

The framework does this by having pipeline components request records from their upstream record sources. See the topic on understanding Forge and downstream record processing for an explanation of this process.

An example Perl manipulator

This Perl manipulator example adds a Record Number property to each record. The example uses the `prepare`, `next_record`, and `finish` methods.

First, the General tab shows the name of the component and that `prepare`, `next_record`, and `finish` methods are checked, which means that each method is defined in the Method Override editor rather than as an external file or class.



Settings shown: Override these methods [✓]; prepare() [✓]; next_record() [✓]; get_records() [ ]; finish() [✓]; Use this Perl class [ ].

Clicking **Edit** for the `prepare()` method displays the Method Override editor, which contains the Perl code shown below. This code makes sure there is only one record source and initializes the record count to zero.

```
# Make sure there is exactly one record source configured
my @source_list = $this->record_sources;

if (scalar(@{ $this->record_sources }) != 1) {
   die("Perl Manipulator ", $this->name,
      " must have exactly one record source.");
}
# Keep the current record number in our context.
$this->context->{RECNO} = 0;
```

Clicking **Edit** for the `next_record()` method displays the Method Override editor which contains the Perl code shown below. This code counts the records processed and tags each record with a property value with that indicates the record number.

```
#Count this record.
++$this->context->{RECNO};
my $rec = $this->record_source(0)->next_record;

# Careful: $rec will be undef if there are no more records.
if ($rec) {
   my $pval = new EDF::PVal("Record Number", $this->context->{RECNO});
   $rec->add_pvals($pval);
}
return $rec;
```

Clicking **Edit** for the `finish()` method displays the Method Override editor, which contains the Perl code shown below. This code prints the number of records processed using the `print_info` method.

```
# Print the number of records processed.
EDF::print_info("Perl Manipulator ".
   $this->name, " processed ".
      $this->context->{RECNO}.
      " records.");
```

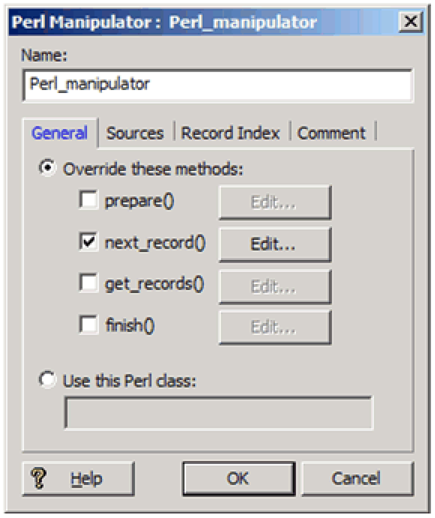
Additional use case examples

This section contains additional use case examples.

Remove a property from each record

This Perl manipulator example removes the Password property from each record as it is processed.

First, the General tab shows the name of the component and that the `next_record` method is checked, which means that `next_record` is defined in the Method Override editor rather than as an external file or class.



Clicking **Edit** for the **next_record()** method displays the Method Override editor which contains the Perl code shown below. This code removes the Password property from each record.

```
# Get the next input record.
my $rec = $this->record_source(0)->next_record;

# Careful: $rec will be undef if there are no more records
if (!$rec) { return undef; }

# There are two ways to do this: the slow way, which is to walk over
# the array of PVals and splice out each one that is named 'Password';
# and the fast way, which is to grep all of them out and re-assign the
# array. We'll do it the fast way.

@{ $rec->pvals } = grep { $_->name ne 'Password' } @{ $rec->pvals };

return $rec;
```

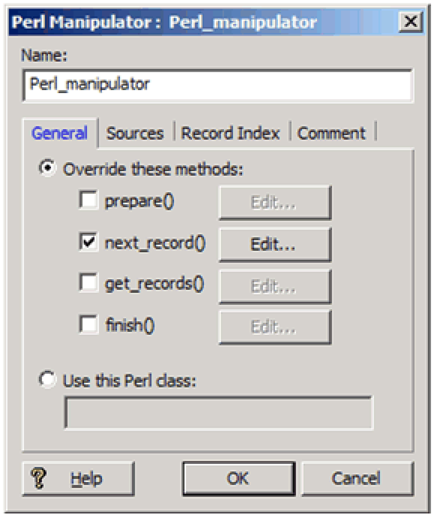
Note: This example could also include a prepare method to perform the same record source validation as in the introduction's example Perl manipulator.

Reformat a property on each record

This example reformats the Price property on each record.

Assume the Price value has no dollar sign. The Perl manipulator replaces the Price value with a number that has a leading dollar sign and exactly two decimal places.

First, the General tab shows the name of the component and that the `next_record` method is checked, which means that `next_record` is defined in the Method Override editor rather than as an external file or class.



Name: Perl_manipulator

(● Override these methods: ○ Use this Perl class:)

- [ ] prepare()
- [✓] next_record()
- [ ] get_records()
- [ ] finish()

Clicking **Edit** for the **next_record()** method displays the Method Override editor which contains the Perl code shown below. This code reformats the Price property on each record.

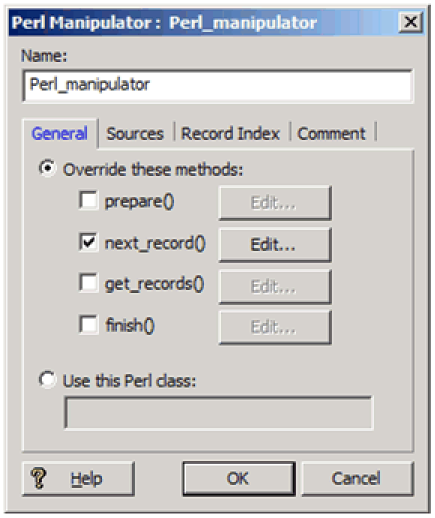
```
# Get the next input record.
my $rec = $this->record_source(0)->next_record;

# Careful: $rec will be undef if there are no more records.
if (!$rec) { return undef; }

# Pull the PVals named 'Price' out of the list of PVals.
my @prices = grep { $_->name eq 'Price' } @{ $rec->pvals };

# If there isn't exactly one price property, print a warning.
if (scalar(@prices) != 1) {
   EDF::print_warning("Expected 1 PVal named Price; found " .
   scalar(@prices));
}
# Reformat all the Price properties. This works even if
# there are 0 properties.
foreach my $pval ( @prices ) {
   # $pval is a reference the same PVal that is on the
   # record, so changing the value here changes it on the record.
   $pval->value = sprintf("\$%.2f", $pval->value);
}
return $rec;
```

Note: This example could also include a `prepare` method to perform the same record source validation as in the introduction's example Perl manipulator.

Remove records with a particular property

This example removes records with the property named Delete This Record.

First, the **General** tab shows the name of the component and that the `next_record` method is checked, which means that `next_record` is defined in the **Method Override editor** rather than as an external file or class.



Name: Perl_manipulator

Override these methods: [✓]
- prepare()
- [✓] next_record()
- get_records()
- finish()

Use this Perl class:

Clicking **Edit** for the `next_record()` method displays the **Method Override editor**, which contains the Perl code shown below. This code removes records with the property named Delete This Record.

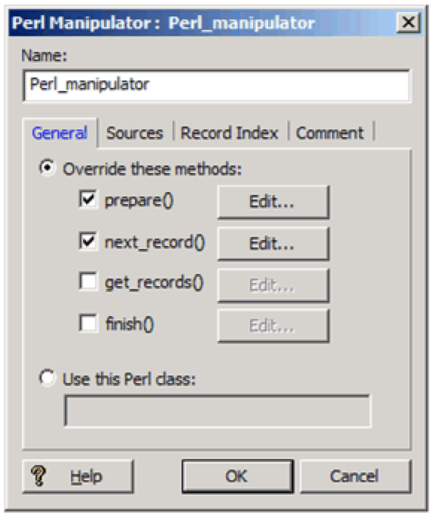
```
# Here's the trick: as long as the record has at least one PVal named
# "Delete This Record", we don't return it; instead, we skip it by
# looping back up and fetching the next record. This effectively
# removes (by skipping) records with PVals named "Delete This Record".
my $rec;
my $skip;

do {
   $rec = $this->record_source(0)->next_record;
   # Careful: $rec will be undef if there are no more records
   return undef unless $rec;
   # Find all the pvals named "Delete This Record"
   my @pvals = grep { $_->name eq "Delete This Record" } @{ $rec->pvals };

   # If there's at least one, skip this record.
   skip = (scalar(@pvals) > 0);
} while ($skip);

# If you want to do additional work, do it here.
return $rec;
```

Note: This example could also include a `prepare` method to perform the same record source validation as in the introduction's example Perl manipulator.

Perform a left join on records from two record sources

This example performs a left join on records from two record sources.

First, the **General** tab shows the name of the component and that both prepare and `next_record` methods are checked, which means that each method is defined in the **Method Override** editor rather than as an external file or class.



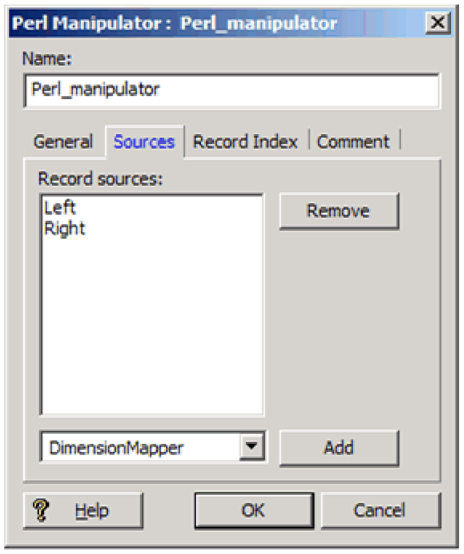
Clicking **Edit** for the `prepare` method displays the **Method Override** editor, which contains the Perl code shown below. This code makes sure there are exactly two record sources.

```
# Make sure we have exactly two record sources
if (scalar(@{ $this->record_sources }) != 2) {
   die("Perl Manipulator ", $this->name,
   " must have exactly two record sources.");
}

# Make sure the record sources are named "Left" and "Right"
# This name is arbitrary, and only done to demonstrate
# retrieving record sources by name.
if (!$this->record_source("Left")) {
   die("Perl Manipulator ", $this->name,
   " must have a record source named 'Left'.");
}

if (!$this->record_source("Right")) {
   die("Perl Manipulator ", $this->name,
   " must have a record source named 'Right'.");
}

# Keep the canonical record key in our context.
# This key has two columns. The first column is the values
# of the property "KeyProp". The second column is the values
# from the dimension with ID 1000.

my $key = new EDF::RecordKey;
$key->add_columns(new EDF::KeyColumn(EDF::KeyColumn::PVAL, "KeyProp"));
$key->add_columns(new EDF::KeyColumn(EDF::KeyColumn::DVAL, 1000));
$this->context->{KEY} = $key;
```

Clicking **Edit** for the `next_record()` method displays the **Method Override** editor, which contains the Perl code shown below. This code joins the two record sources.

```
my $left_src = $this->record_source('Left');
my $right_src = $this->record_source('Right');
my $left_rec;
my @right_recs;
$left_rec = $left_src->next_record;
if (!$left_rec) { return undef; } # End of input

# This makes a copy of the canonical key and fills it in.
my $key = $left_rec->key($this->context->{KEY});

# Find matching records on the right-hand side.
@right_recs = $right_src->get_records($key);

# Move PVals and DVals from the right-hand records onto the
# left-hand records.

foreach my $right_rec ( @right_recs ) {
   # Don't move KeyProp over.
   my @pvals = grep { $_->name ne "KeyProp" } @{ $right_rec->pvals };
   $left_rec->add_pvals(@pvals);
   # Don't move DVals from dimension 1000 over.
   my @dvals = grep { $_->dimension_id != 1000 } @{ $right_rec->dvals };
   $left_rec->add_dvals(@dvals);
}
return $left_rec;
```

Clicking the **Sources** tab of the Perl Manipulator editor shows that both Left and Right record sources are available.



Note: A Perl manipulator causes Forge to fail when the Perl manipulator is on the left side of a left join and no record cache adapter is used. Forge returns the following error:

```
---FATAL ERROR---
FTL: [LoadData]: (RecordAdapter.cpp:566) Can't call nextRecordInput at end-
```

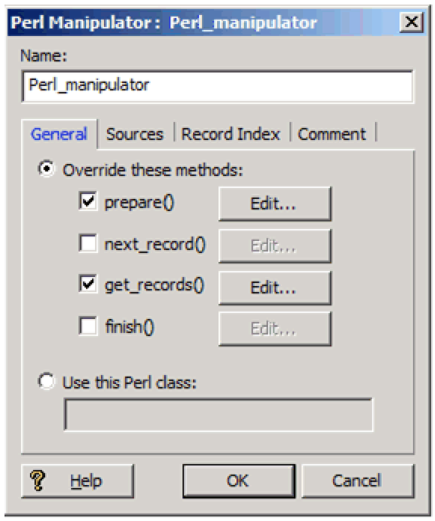
```
of-stream
on RecordAdapter 'LoadData'
```

To work around this issue, add a record cache adapter after the Perl manipulator. See the *Endeca Developer Studio Help* for details about using a record cache.

Retrieve records matching a key from any number of record sources

This example retrieves records from any number of record sources that match a key.

First, the **General** tab shows the name of the component and that the `prepare` and `get_records()` methods are checked, which means that they are defined in the **Method Override** editor rather than as an external file or class.


Perl Manipulator : Perl_manipulator

Name: Perl_manipulator

Override these methods: [✓ prepare()] next_record() [✓ get_records()] finish()

Use this Perl class:

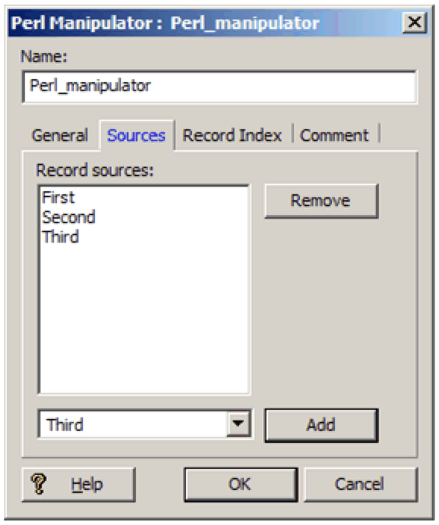
Clicking **Edit** for the `prepare()` method displays the **Method Override** editor which contains the Perl code shown below. This code sets a flag for later manipulation on the first call of `get_records()`.

```
# We want to do special processing on the first call to
# GET_RECORDS.
# Setting this context flag will let us determine later
# what to do for the first call.
$this->context->{FIRST} = 1;
```

Clicking **Edit** for the `get_records()` method displays the **Method Override** editor, which contains the Perl code shown below. This code retrieves records from all the record sources available on the **Sources** tab of the **Perl Manipulator** editor.

```
my $key = shift;
my @recs;

# Things to do on the first call only.
if ($this->context->{FIRST}) {

# Not the first call any more; reset the flag.
$this->context->{FIRST} = 0;

# Build a hash of the names of PVals that are in the key.
$this->context->{PVAL_KEYS} = \{ };
# Build a hash of the dimension IDs of that are in the key.
$this->context->{DVAL_KEYS} = \{ };
   foreach $col ( @{ $key->columns } ) {
      if ($col->type eq PVAL) {
         $this->context->{PVALS}->{$col->id} = 1;
         }
      else {
         $this->context->{DVALS}->{$col->id} = 1;
         }
      }
   }
# Do the rest on each call to get_records.
foreach my $src ( $this->record_sources ) {
my @fetched = $src->get_records($key);

# Remove the PVals that are in the key.  This is done on the
# assumption that we're on the right-hand side of a join, so
# the key PVals are already on the record used to make the key
# that was passed to us. Leaving these PVals on the key would
# create duplicates that we'd have to remove later.
   foreach my $rec ( @fetched ) {
   # Remove PVals that are in the key.
   @{ $rec->pvals } =
      grep { !$this->context->{PVALS}->{$_->name} }
         @{ $rec->pvals };
   # Remove DVals that are in the key.
   @{ $rec->dvals } =
      grep { !$this->context->{DVALS}->{$_->dimension_id} }
         @{ $rec->dvals };
   }
push(@recs, @fetched);
}
return @recs;
```

Clicking the **Sources** tab of the **Perl Manipulator** editor shows that the First, Second, and Third record sources are available.


Perl Manipulator : Perl_manipulator

Name: Perl_manipulator

Record sources:
First
Second
Third

Third

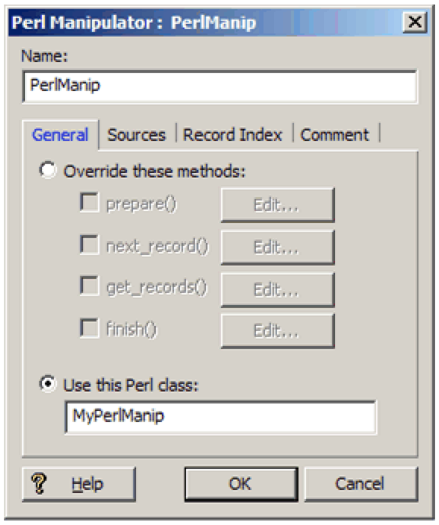
Process records using a subclass

This example uses a subclass of `EDF::Manipulator` to add a property to each record in the same way as in the first example Perl manipulator. This approach provides an alternative to writing your Perl code in the **Method Override** editor or in a Perl file.

Note: The Perl class must be located on the machine running Forge. It is convenient to locate the PM file in the in the same location as other Perl modules for Endeca (`ENDECA_ROOT\lib\perl`). Placing your PM file in `ENDECA_ROOT\lib\perl` does not require any additional configuration for Forge to locate it. However, if you upgrade Forge, you will have to copy this file to another location and copy it back in after upgrading.

If you place the file in a location other than `ENDECA_ROOT\lib\perl`, you must modify Perl's library search path to include the path to the PM file. You can modify the path by either modifying your *PERLLIB* environment variable or by running Forge with the `--perllib` command line option and providing the path as an argument.

The Perl manipulator component is configured to access a Perl class as shown:


Perl Manipulator : PerlManip
Name: PerlManip

Use this Perl class: MyPerlManip

The contents of the `MyPerlManip` class are as follows:

```
package MyPerlManip;
use EDF::Manipulator;
@ISA = qw(EDF::Manipulator);
sub new
{
   my $proto = shift;
   my $class = ref($proto) || $proto;
   my $this = {};
   bless($this, $class);
   $this->SUPER::init(@_);
   return $this;
}
sub prepare
{
   # Make sure there is exactly one record source configured
   my @source_list = $this->record_sources;
   if (scalar(@{ $this->record_sources }) != 1) {
      die("Perl Manipulator ", $this->name,
      " must have exactly one record source.");
   }
   $this->{RECNO} = 0;
}
sub next_record
{
   my $this = shift;
   ++$this->{RECNO};
   my $rec = $this->record_source(0)->next_record;
   if ($rec) {
      my $pval = new EDF::PVal("Record Number", $this->{RECNO});
      $rec->add_pvals($pval);
   }
return $rec;
}
```

Add a geocode property to a record

An application may include geocode properties in its records to enable record sorting by distance from a given reference point.

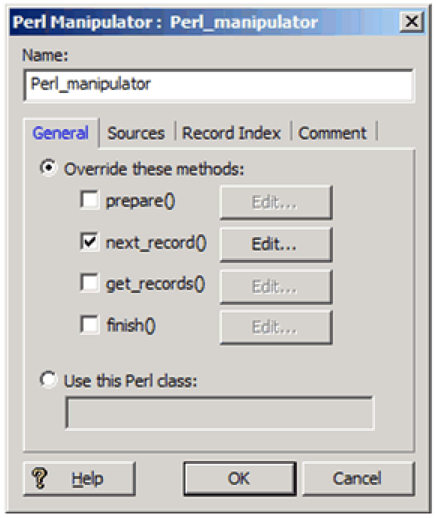
The indexer requires geocode data in the form `d,d` where each `d` is a double-precision floating-point value:

- The first `d` is the latitude of the location in whole and fractional degrees. Positive values indicate north latitude and negative values indicate south latitude.
- The second `d` is the longitude of the location in whole and fractional degrees. Positive values indicate east longitude, and negative values indicate west longitude.

For example, Endeca's main office is located at 42.365615 north latitude, 71.075647 west longitude. This geocode should be supplied to the indexer as "42.365615,-71.075647". If the input data is not available in this format, it can be assembled from separate properties using EDF Perl.

This example creates a Location property by concatenating the values of a record's Latitude property and its Longitude property.

First, the **General** tab shows the name of the component and that the `next_record` method is checked, which means that `next_record` is defined in the **Method Override** editor rather than as an external file or class.



Name: Perl_manipulator

[✓ Override these methods:]

- prepare()
- [✓] next_record()
- get_records()
- finish()

Use this Perl class:

Clicking **Edit** for the `next_record` method shows the Perl code. This code creates a Location property and adds it to each record that contains Latitude and Longitude properties.

Note: The contents of the **Method Override** editor are as follows:

```
#Get the next record from the first record source.
my $rec = $this->record_source(0)->next_record;
return undef unless $rec;
#Return an array of property values from the record.
my @pvals = @{$rec->pvals};

#Return the value of the Latitude property.
my @lat = grep {$_->name eq "Latitude"} @{$rec->pvals};

#Return the value of the Longitude property.
my @long = grep {$_->name eq "Longitude"} @{$rec->pvals};

#Exit if there is more than one Latitude property.
if (scalar (@lat) !=1) {
   die("Perl Manipulator ", $this->name,
   " must have exactly one Latitude property.");
}

#Exit if there is more than one Longitude property.
if (scalar (@long) !=1) {
   die("Perl Manipulator ", $this->name,
   " must have exactly one Longitude property.");
}

#Concatenate Latitude and Longitude into Location.
my $loc = $lat[0]->value . "," . $long[0]->value;

#Add new Location property to record.
my $pval = new EDF::PVal("Location", $loc);
$rec->add_pvals($pval);

return $rec;
```

Note: This example could also include a `prepare` method to perform the same record source validation as in the introduction's example Perl manipulator.

Chapter 2

EDF::DVal class

This section describes the `EDF::DVal` class.

EDF::DVal class overview

An `EDF::DVal` object represents a dimension value tagged to a record or contained in an `EDF::RecordKey`.

Constructor

The new method constructs a new `EDF::DVal` object. The method takes up to two optional input arguments, the dimension ID, and the DVal ID. For example:

```
my $empty_dval = new EDF::DVal;
my $full_dval = new EDF::DVal(1000, 1234);</p>
```

Methods

- `dimension_id`
- `id`

EDF::DVal::dimension_id method

The `dimension_id` method returns a scalar value representing the ID of the dimension that contains the DVal.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $dval = $rec->dvals->[0];
if ($dval->dimension_id == 1000) {
   $dval->dimension_id = 2000; # Swap dimensions
}
```

Input Arguments

None.

Return Values

A scalar value representing the ID of the dimension that contains the DVal.

EDF::DVal::id method

The `id` method returns a scalar value representing the ID of the DVal.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $dval = $rec->dvals->[0];
if ($dval->id == 1234) {
   $dval->id = 4321; # Change all 1234's into 4321's
}
```

Input Arguments

None.

Return Values

A scalar value representing the ID of the DVal.

Chapter 3

EDF::Manipulator class

his section describes the `EDF::Manipulator` class.

EDF::Manipulator class overview

Each Perl manipulator that Forge runs as part of your pipeline is an instance of the `EDF::Manipulator` class.

The `EDF::Manipulator` class varies from other classes in the EDF namespace because you, the pipeline developer, write the body of the overrideable methods for an `EDF::Manipulator`. The overrideable methods are `prepare`, `get_records`, `next_records`, and `finish`.

Note: For convenience, the local variable `$this` is already initialized to reference the `EDF::Manipulator` instance when your code is evaluated. This initialization is approximately equivalent to the code:

```
package EDF::Manipulator;
sub next_record($) {
   my $this = shift;
   # Your code goes here
}
```

Constructor

There is no constructor method for this class. Forge constructs an `EDF::Manipulator` object at runtime when Forge encounters a Perl manipulator component in your pipeline.

Methods

- `context`
- `finish`
- `get_records`
- `name`
- `next_record`
- `prepare`
- `record_source`
- `record_sources`

EDF::Manipulator::context method

The `context` method stores information that needs to persist across method calls and that needs to be restricted to a specific Perl manipulator.

For example, you might use this method to store the number of records processed, a key used for lookups, and so forth. This method is useful when you have multiple Perl manipulators that each store information, such the number of records or a particular key, and that information must not be shared among them. The information stored by the context method is not global— each Perl manipulator in a pipeline has its own context.

Example Usage

```
# Recall $this is initialized with the EDF::Manipulator instance.
$this->context->{RECNO} += 1;
```

Input Arguments

None.

Return Values

A reference to a hash.

EDF::Manipulator::finish method

The `finish` method contains your custom-written Perl code that you want Forge to call after record processing is complete. The code you provide here becomes the body of the `finish` method.

Calls to `next_record` or `get_records` methods result in an error. To signal a failure during finish, call Perl's `die()` function. See the Introduction's example Perl manipulator for a sample usage of this method.

Input Arguments

None.

Return Values

None.

EDF::Manipulator::get_records method

The `get_records` method contains your custom-written Perl code that Forge calls to retrieve all the records matching an `EDF::RecordKey` argument you specify.

The code you provide here becomes the body of the `get_records` method. Although you write the Perl code to determine how Forge implements `get_records`, you do not call this method. Forge calls this method when the component downstream of the Perl manipulator requests records. The `get_records` method returns an array of records to the requesting pipeline component. Like the `next_record` method, `get_records` has access to any record sources specified on the **Sources** tab of your Perl Manipulator editor.

See the example that retrieves records matching a key for a sample usage.

Input Arguments

Provide an `EDF::RecordKey` argument to identify records.

Return Values

An array of records matching the provided `EDF::RecordKey`. This array is passed back to the requesting pipeline component. Returning an empty list or undef does not signal the end of processing. An `undef` or an empty list indicates there are no records that match the key provided.

EDF::Manipulator::name method

The `name` method returns the value of the Name field that you specified in the **Perl Manipulator** editor.

The local variable `$this` is initialized to contain the `EDF::Manipulator` instance for the Perl manipulator. Such initialization means calling `$this->name;` returns the name of the `EDF::Manip¬ulator` instance.

Example Usage

```
EDF::print_info("This Perl manipulator is named ". $this->name);
```

Input Arguments

None.

Return Values

The name of the Perl manipulator.

EDF::Manipulator::next_record method

The `next_record` method contains your custom-written Perl code that returns the next record from this Perl manipulator during record processing.

Although you write the Perl code to determine how the next record should be returned to Forge, you do not call this method. Forge calls this method when a component downstream of the Perl manipulator requests the next record. The `next_record` method returns a single record per invocation. Like the `get_records` method, `next_record` has access to any record sources specified on the **Sources** tab of the Perl manipulator.

See the example that reformats a property on each record for a sample usage.

Input Arguments

None.

Return Values

A single record, which is passed to the requesting pipeline component. To signal the end of records to be processed, return the `undef` value.

EDF::Manipulator::prepare method

The `prepare` method contains custom-written Perl code that you want Forge to call before record processing starts.

This code typically performs pre-processing tasks such as checking data sources, initializing record counts, and so on. The code you provide here becomes the body of the prepare method. Calling either `next_record` or `get_records` from prepare results in an error. To signal a failure during prepare, call Perl's `die()` function. Also see the `Manipulator::finish` method.

See the Introduction's example Perl manipulator for a sample usage of this method.

Input Arguments

None.

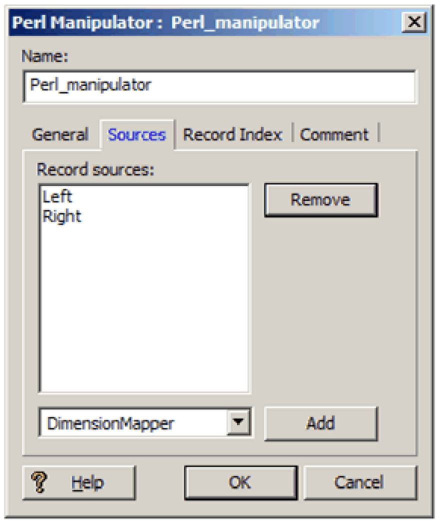
Return Values

None.

EDF::Manipulator::record_source method

The `record_source` method provides the Perl manipulator access to any record source specified on the **Sources** tab of the Perl Manipulator editor.

Sources can be identified by either name or position number. The number of a record source is determined by the position in which it appears in the **Sources** tab of the **Perl Manipulator** editor beginning from the index point 0. In this example, record source number 0 is named "Left," and record source number 1 is named "Right."



Name: Perl_manipulator

Record sources:
Left
Right

DimensionMapper

Example Usage

This example accesses the record source identified by name:

```
my $rec = $this->record_source("Left")->next_record;
```

This example accesses the record source identified by position:

```
my $rec = $this->record_source(0)->next_record;
```

Input Arguments

Either the name or position number of the desired record source.

Return Values

An `EDF::RecordSource` object or `undef` if the indicated source does not exist.

EDF::Manipulator::record_sources method

The `record_sources` method provides a Perl manipulator access to all the record sources specified on the **Sources** tab of the **Perl Manipulator** editor.

See `record_source` to get a single record source.

Example Usage

```
# Make sure we have exactly two record sources
if (scalar(@{ $this->record_sources }) != 2) {
   die("Perl manipulator ", $this->name,
   " must have exactly two record sources.");
```

Input Arguments

None.

Return Values

An array of `EDF::RecordSource` objects.

Chapter 4

EDF::PVal class

This section describes the `EDF::PVal` class.

EDF::PVal class overview

An `EDF::PVal` object represents a property value on a record or an `EDF::RecordKey`.

Constructor

The new method creates a new `EDF::PVal` object. The method takes two optional input arguments, the name and value of the `PVal`. For example:

```
my $empty_pval = new EDF::PVal;
my $full_pval = new EDF::PVal("Name", "Value");
```

Methods

- `name`
- `value`

EDF::PVal::name method

The name method returns a scalar value representing the name of the `EDF::PVal`. Manipulating the value changes the `EDF::PVal` object.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $pval = $rec->pvals->[0];
if ($pval->name =~ /^Endeca\./) {
   $pval->name = "Internal Property.";
}
```

Input Arguments

None.

Return Values

A scalar value representing the name of the `EDF::PVal`.

EDF::PVal::value method

The value method returns a scalar value representing the value of the `EDF::PVal` object. Manipulating the value changes the `EDF::PVal` object.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $pval = $rec->pvals->[0];
if ($pval->value =~ /^file:/) {
   $pval->value = "File Path";
}
```

Input Arguments

None.

Return Values

A scalar value representing the value of the `EDF::PVal`.

Chapter 5
EDF::RecordKey class

This section describes the `EDF::RecordKey` class.

EDF::RecordKey class overview

An `EDF::RecordKey` contains the information necessary to identify records for record selection and joins. An `EDF::RecordKey` contains one or more `EDF::KeyColumn` objects that represent values for comparison.

Constructor

The new method constructs a new `EDF::RecordKey`. The constructor takes no input arguments. For example:

```
my $key = new EDF::RecordKey;
```

Methods

- `add_columns`
- `clone`
- `columns`
- `equals`

EDF::RecordKey::add_columns method

The `add_columns` method adds one or more `EDF::KeyColumn` objects to an `EDF::RecordKey`.

Example Usage

```
my $key = shift;
$key->add_columns(new EDF::KeyColumn);
```

Input Arguments

A list of `EDF::KeyColumn` objects.

Return Values

None.

EDF::RecordKey::clone method

The `clone` method takes an `EDF::RecordKey` and clones it according to the `EDF::PVal` and `EDF::DVal` contents of the `EDF::RecordKey`. The `EDF::PVal` and `EDF::DVal` values in the `RecordKey` are not shared between the original and the copy.

By contrast, `EDF:Record:key` creates a copy where `EDF::PVal` and `EDF::DVal` values are shared between the original and the copy.

Example Usage

```
my $key = shift;
my $copy = $key->clone;
```

Input Arguments

An `EDF::RecordKey` object.

Return Values

A new `EDF::RecordKey` object.

EDF::RecordKey::columns method

The `columns` method returns a reference to the array of columns in the `EDF::RecordKey`. Manipulating the referenced array changes the `EDF::RecordKey`.

Example Usage

```
my $key = shift;
if (scalar @{ $key->columns } > 0) {
   @{ $key->columns } = ( ); # Remove all columns
}
```

Input Arguments

None.

Return Values

A reference to the array of columns in the `EDF::RecordKey`.

EDF::RecordKey::equals method

The `equals` method takes one argument, another `EDF::RecordKey`, and returns 1 if the second key is the same as the first or 0 if it is not.

Example Usage

```
my $key = shift;
my $copy = $key->clone;
if (!$key->equals($copy)) {
   die("Something is awry.");
}
```

Input Arguments

An `EDF::RecordKey` object.

Return Values

1 if the record keys are equal or 0 if they are not.

Chapter 6

EDF::KeyColumn class

This section describes the EDF::KeyColumn class.

EDF::KeyColumn class overview

An EDF::KeyColumn object represents a value for comparison within an EDF::RecordKey. Each column has a type, an ID, and zero or more values that are either EDF::DVal or EDF::PVal objects.

Constructor

The new method constructs a new EDF::KeyColumn object. The method takes two input arguments, a type and an ID. See the type and id methods below for more information. For example:

```
my $col = new EDF::KeyColumn(PVAL, "Color");
```

Methods

- add_values
- id
- type
- values

Constants

- PVAL
- DVAL

EDF::KeyColumn::add_values method

The add_values method adds one or more values to the column. You can add either property values or dimension values; however, the type of the value must match the columntype.

You can use the KeyColumn::type method to check whether EDF::KeyColumn values are type EDF::KeyColumn::PVAL or EDF::KeyColumn::DVAL.

Example Usage

```
my $key = new EDF::RecordKey;
my $col = new EDF::KeyColumn;
$key->add_columns($col);
$col->type = PVAL;
my @pvals;
for my $i ( 1..10 ) {
   push(@pvals, new EDF::PVal("Number", $i));
}
$col->add_values(@pvals);
$col = new EDF::KeyColumn;
$col->type = DVAL;
$key->add_columns($col);
my @dvals;
for my $i ( 1001..1010 ) {
   push(@dvals, new EDF::DVal(1000, $i));
}
$key->add_values(@dvals);
```

Input Arguments

Either `EDF::PVal` or `EDF::DVal` objects depending on the column type.

Return Values

None.

EDF::KeyColumn::id method

The `id` method returns a scalar value representing the ID of the column.

This value is either a property name, if the column type is `KeyColumn::PVAL`, or a dimension ID if the column type is `KeyColumn::DVAL`. Manipulating the value changes the `EDF::KeyColumn` object.

Example Usage

```
my $pcol = new EDF::KeyColumn;
$pcol->type = EDF::KeyColumn::PVAL;
$pcol->id = "Color";
my $dcol = new EDF::KeyColumn;
$dcol->type = EDF::KeyColumn::DVAL;
$dcol->id = 1000;
```

Input Arguments

None.

Return Values

Either a property name or a dimension ID.

EDF::KeyColumn::type method

The `type` method returns a scalar value representing the type of the column.

Example Usage

This example creates a new `KeyColumn` object and assigns it the type `PVAL`.

```
my $col = new EDF::KeyColumn;
$col->type = EDF::KeyColumn::PVAL;
```

Input Arguments

None.

Return Values

Either an `EDF::KeyColumn::PVAL` constant or an `EDF::KeyColumn::DVAL` constant.

EDF::KeyColumn::values method

The `values` method returns a reference to the array of values in the `EDF::KeyColumn` object.

Each entry in the array is either an `EDF::PVal` or an `EDF::DVal`. Manipulating the referenced array changes the `EDF::KeyColumn` object.

Example Usage

```
my $key = shift;
for $col ( @{ $key->columns }) {
   if ($col->type eq PVAL) {
      my @pvals = @{ $col->values };
   }
   else {
      my @dvals = @{ $col->values };
   }
}
```

Input Arguments

None.

Return Values

A reference to the array of values in the column.

EDF::KeyColumn::DVAL constant

The `DVAL` constant indicates that the column type contains `EDF::DVal` values. If you need to compare this value to others, use Perl's string comparison operators `eq` or `ne`.

Example Usage

This example creates a new `KeyColumn` object of type `DVAL`.

```
my $col = new EDF::KeyColumn;
$col->type = EDF::KeyColumn::DVAL;
```

This example shows a comparison using `eq`:

```
if ($col->type eq EDF::KeyColumn::DVAL) {...}
```

EDF::KeyColumn::PVAL constant

The `PVAL` constant indicates that the column type contains `EDF::PVal` objects. If you need to compare this value to others, use Perl's string comparison operators `eq` or `ne`.

Example Usage

This example creates a new `EDF::KeyColumn` object of type `PVAL`:

```
my $col = new EDF::KeyColumn;
$col->type = EDF::KeyColumn::PVAL;
```

This example shows a comparison using `eq`:

```
if ($col->type eq EDF::KeyColumn::PVAL) {...}
```

Chapter 7

EDF::Record class

This section describes the `EDF::Record` class.

EDF::Record class overview

An `EDF::Record` object describes a record. Records have a list of property values and a list of dimension values.

Constructor

The new method creates a new `EDF::Record` object. The method takes no arguments. For example:

```
my $rec = new EDF::Record;
```

Methods

- `add_dvals`
- `add_pvals`
- `clone`
- `dvals`
- `key`
- `pvals`

EDF::Record::add_dvals method

The `add_dvals` method adds a list of `EDF::DVal` objects to the record.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $dval = new EDF::DVal(1, 1);
$rec->add_dvals($dval);
my @newvals;
foreach $n ( 1..10 ) {
   push(@newvals, new EDF::DVal(1, $n));
}
$rec->add_dvals(@newvals);
```

Input Arguments

A list of `EDF::DVal` objects.

Return Values

None.

EDF::Record::add_pvals method

The `add_pvals` method adds a single `EDF::PVal` or a list of `EDF::PVals` to the record.

This method is synonymous to the following code: `push(@{$rec->pvals},@newvals);`

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $pval = new EDF::PVal("Number", 1);
$rec->add_pvals($pval);
my @newvals;
foreach $n ( 1..10 ) {
   push(@newvals, new EDF::PVal("Number", $n));
}
$rec->add_pvals(@newvals);
```

Input Arguments

A single `EDF::PVal` or a list of `EDF::PVal` objects.

Return Values

None.

EDF::Record::clone method

The `clone` method creates a copy of the record including the `EDF::PVal` and `EDF::DVal` items on the record. This is useful if you want to make a copy of a record to modify, while retaining a copy of the original.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
# $clone has copies of $rec's PVals
my $clone = $rec->clone;
# Rename the first PVal on $clone to "Wilma" -
# the first PVal on $rec is left untouched.
$clone->pvals->[0]->name = "Wilma";
```

Input Arguments

An `EDF::Record` object.

Return Values

A new `EDF::Record` object.

EDF::Record::dvals method

The `dvals` method returns an array of the dimension values tagged to a record. Manipulating the referenced array changes the `EDF::Record` object.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $dvals = $rec->dvals;
# Add a constant DVal to the record
push(@{$dvals}, new EDF::DVal(1,1));
```

Input Arguments

None.

Return Values

An array of the dimension values tagged to the record.

EDF::Record::key method

The key method takes an `EDF::RecordKey`, clones it according to the contents of the `EDF::Record`, and fills the dimensions of the cloned key in with the appropriate `EDF::PVal` and `EDF::DVal` values. The resulting copy has `EDF::PVal` and `EDF::DVal` values that are shared between the original and the copy. See also `EDF::RecordKey:clone`.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $col = new EDF::KeyColumn(EDF::KeyColumn::PVAL, "KeyProp");
my $empty_key = new EDF::RecordKey;
$empty_key->add_columns($col);
my $filled_in_key = $rec->key($empty_key);
```

Input Arguments

An `EDF::RecordKey` object.

Return Values

A new `EDF::RecordKey` object.

EDF::Record::pvals method

The `pvals` method returns a reference to the array of property values (`EDF::PVal` objects) on the record. Manipulating the referenced array changes the `EDF::Record` object.

Example Usage

```
my $rec = $this->record_source(0)->next_record;
my $pvals = $rec->pvals;
# Change the name of the first PVal on the record
# to "Wilma"
$pvals->[0]->name = "Wilma";
# Remove all PVals from the record
@{ $pvals } = ( );
```

Input Arguments

None.

Return Values

A reference to the array of `EDF::PVal` objects (property values) on the record.

Chapter 8

EDF::RecordSource class

This section describes the `EDF::RecordSource` class.

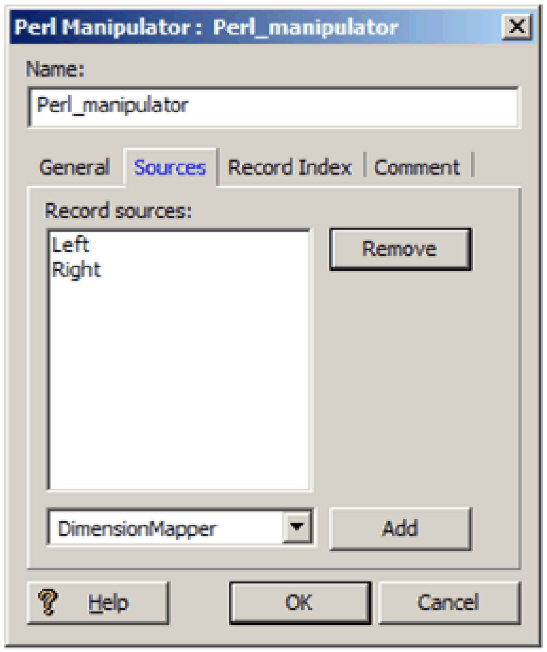
EDF::RecordSource class overview

An `EDF::RecordSource` object represents the record source specified on the **Sources** tab of the Perl Manipulator editor.

The data in this object becomes the record input that the `EDF::Manipulator` modifies. An `EDF::RecordSource` has a name and a number, and can be addressed by either. You can retrieve records from an `EDF::RecordSource` using the `next_record` and `get_records` methods.

Note: The number of a record source is determined by the position in which it appears in the **Sources** tab of the **Perl Manipulator** editor beginning from the index point 0.

In this example, record source number 0 is named "Left," and record source number 1 is named "Right."



Constructor

There is no constructor method for this class. Calling the `EDF::Manipulator::record_source` method returns an `EDF::RecordSource` object.

Methods

- `get_records`
- `name`
- `next_record`
- `number`

EDF::RecordSource::get_records method

The `get_records` method returns the list of records matching a specified `EDF::RecordKey`.

Note that different pipeline components that act as record sources handle the `get_records` call in different ways. For example, a record cache retains copies of all records, and can return multiple copies if the same request is made repeatedly. Most other components do not retain copies. They return a matching record the first time a given request is made and discard non-matching records between request.

Example Usage

```
my $key = shift;
my $src = $this->record_source(0);
my @recs = $src->get_records($key);
```

Input Arguments

An `EDF::RecordKey` object.

Return Values

The list of records matching a given `EDF::RecordKey`.

EDF::RecordSource::name method

The `name` method returns the name of a record source as specified on the **Sources** tab of the **Perl Manipulator** editor.

Example Usage

```
my $src = $this->record_source(0);
EDF::print_info("Record source 0 is ".
   $src->name);
```

Input Arguments

None.

Return Values

The name of the record source.

EDF::RecordSource::next_record method

The `next_record` method returns the next record from this source, or `undef` if there are no more records available from this source.

Example Usage

```
my $src = $this->record_source(0);
my $rec = $src->next_record;
```

Input Arguments

None.

Return Values

The next record from this source, or `undef` if there are no more records available from this source.

EDF::RecordSource::number method

The `number` method returns the index of the record source in the array of record sources for a particular Perl manipulator.

Note that if the same record source serves as an input to multiple Perl manipulators (for example, a record cache), it may have a different number in each.

Example Usage

```
my $src = $this->record_source('RecordCache');
EDF::print_info("RecordCache is source number ".
$src->number);
```

Input Arguments

None.

Return Values

The index of the record source in the array of record sources for a particular Perl Manipulator.

Chapter 9

Static methods

This section describes the `Static` methods.

Static methods overview

The Static methods can be used in any Forge Perl code to log messages.

Use these methods to log different levels of messages (ERR, INF, WRN): `print_error`, `print_info`, and `print_warning`.

EDF::print_error method

The `EDF::print_error` method prints a message at log level `ERR`.

A terminating new line token (\n) is not required. `EDF::print_error` does not cause Forge to exit. Call Perl's `die()` function to exit Forge. When Forge exits, it exits with an error code.

Example Usage

```
# Make sure we have exactly one record source,
# or print error and stop Forge.
if (scalar(@{ $this->record_sources }) != 1) {
   EDF::print_error ("Perl Manipulator ", $this->name .
   " must have exactly one record source.");
   die("Perl Manipulator ", $this->name,
   " must have exactly one record source.");
}
```

Input Arguments

A string message to print.

Return Values

None.

EDF::print_info method

The `EDF::print_info` method prints a message at log level `INF`.

A terminating new line token (\n) is not required.

Example Usage

```
my $src = $this->record_source('RecordCache');
EDF::print_info("RecordCache is source number " .
$src->number);
```

Input Arguments

A string message to print.

Return Values

None.

EDF::print_warning method

The `EDF::print_warning` method prints a message at log level `WRN`.

A terminating new line token (\n) is not required.

Example Usage

```
# If there isn't exactly one price property, print a warning.
if (scalar(@prices) != 1) {
   EDF::print_warning("Expected 1 PVal named Price; found" .
      scalar(@prices));
}
```

Input Arguments

A string message to print.

Return Values

None.

Index