Build a custom search engine with PHP

Sphinx indexes your content, finds text fast, and provides useful search results

While Google and its ilk are virtually omniscient, the Web's mighty search engines aren't well suited to every site. If your site content is highly specialized or distinctly categorized, use Sphinx and PHP to create a finely tuned local search system.

Share:

Martin Streicher is the Editor-in-Chief of Linux Magazine. Martin earned a Master of Science in Computer Science from Purdue University and has been programming UNIX-like systems since 1986 in the Pascal, C, Perl, Java, and (most recently) Ruby programming languages.

31 July 2007

Also available in Russian Japanese

Develop and deploy your next app on the IBM Bluemix cloud platform.

Start building for free

In the Internet age, people want information packaged like fast food: served instantly, hassle-free, and in bite-size (or is that *byte*-size?) morsels. Indeed, to feed the impatient and hungry masses, even the most modest Web site is now expected to serve a diverse menu of fast-fact formats:

RSS is your pizza guy, bringing fresh-baked data right to your door.

The weblog is your local Chinese take-out, delivering your favorite spicy dish.

The forum is the neighborhood potluck (or perhaps more aptly, the food fight scene in "Animal House").

And search is like all-you-can-eat night at your local cafeteriaplex: Just fill up your plate with anything your heart desires, time and again, so long as your gullet — and your chair — hold up.

Luckily, PHP developers can find a wide variety of RSS, blog, and forum software to create or amend a site. And while Google and others are virtually omniscient and do funnel traffic, the search engines aren't necessarily well suited to every site.

For example, if your Web site offers hundreds of thousands of new and refurbished Porsche parts, Google may turn up your site for a broad search such as "Carrera parts," but it may not yield an accurate result for the more specific "used 1991 Porsche 911 Targa headlight bezel" inquiry.

If your content is highly specialized or your visitors expect your search feature to parallel real-world workflow, it's best to augment the Web's global search engines with a local search system tailored to your site. (See "A needle in a billion haystacks" for more instances of specialized search.)

Discover how to add a fast, capable, open source, and free search engine to a PHP site. Little of the visible Web site is developed here. Instead, the focus is on the components required to deliver effective search results: the database, the index, the search engine, and the PHP application program interface (API).

Visit the great sphinx

To provide a custom search feature for your site, you must have a data source and the ability to search

that source. For a Web application, the data source is commonly a relational database, which has some forms of search built in. (Equality is a simple search operator, as is the SQL operator `LIKE`.) Yet, some searches may be more specialized than the database can perform, or a search may be so complicated that the inherent SQL `JOIN`s are simply too slow.

To speed searches, you may be able to rearrange your tables and, thus, simplify the underlying queries. (Table and SQL query optimization are highly dependent on your schema and engine. Search online to find a vast number of articles and books dedicated to database performance.) Alternatively, you can add a specialized search engine. Which search engine to apply also depends on the form (and quantity) of your data and your budget. Many options are available: You can connect a Google appliance to your network, purchase Endeca or another large-scale commercial search product, or try Lucene. But in many cases, commercial products are overkill or squander an operating budget, and Lucene did not offer a PHP API when this was written in July 2007.

A needle in a billion haystacks

Many sites provide content specific to an industry, vocation, or pastime, such as medicine, law, music, and auto maintenance. Delving into such content may require special tools or training, or it may simply require a singular index to generate relevant and practical results.

Here are some common search scenarios that require a tailor-made search system:

Find all articles about the Stanley Cup written by a Joe Hockey.
Find the latest drivers for the HP LaserJet 3015 All-in-One printer.
Find the footage of Dinosaur Jr.'s, performance from the Late Show With David Letterman.

As an alternative, consider Sphinx, which is an open source and free (as in speech and beer) search engine designed to search text extremely quickly. For instance, on a live database of nearly 300,000 rows of five indexed columns, where each column contains about 15 words, Sphinx can yield a result for an "any of these words" search in 1/100th of a second (on a 2-GHz AMD Opteron processor with 1 GB of RAM running Debian Linux® Sarge).

Sphinx has a raft of features, including:

- It can index any data you can represent as a string.
- It can index the same data in different ways. With multiple indexes, each tuned for a specific purpose, you can choose the most appropriate index to optimize search results.
- It can associate attributes with each piece of indexed data. You can then use one or more of the attributes to further filter search results.
- It supports morphology, so a search for the word "cats" also finds the root word "cat."
- You can distribute a Sphinx index among many machines, providing failover.
- It can create indexes of word prefixes of arbitrary length and indexes of infix substrings of varying lengths. For instance, a part number may be 10 characters wide. The prefix index would match against all possible substrings anchored at the start of the string. The infix index would match substrings anywhere within the string.
- You can run it as a storage engine within MySQL V5, alleviating the need for yet another daemon, which is often viewed as an additional point of failure.

You can find a complete list of features online and in the README file distributed with the Sphinx source code. The Sphinx Web site also lists several projects that have deployed Sphinx.

Sphinx is written in C++, builds with the GNU compilers, supports 64-bit on capable platforms, and runs on Linux, UNIX®, Microsoft® Windows®, and Mac OS X. Building Sphinx is simple: Download and extract the code, then run the command `./configure && make && make install`.

By default, Sphinx utilities are installed in /usr/local/bin/, and the configuration file for all the Sphinx components is /usr/local/etc/sphinx.conf.

Sphinx has three components: an index generator, a search engine, and a command-line search utility:

The index generator is called *indexer*. It queries your database, indexes each column in each row of the result, and ties each index entry to the row's primary key.

The search engine is a daemon called *searchd*. The daemon receives search terms and other parameters, scours one or more indices, and returns a result. If a match is made, searchd returns an array of primary keys. Given those keys, an application can run a query against the associated database to find the complete records that comprise the match. Searchd communicates to applications through a socket connection on port 3312.

The handy *search* utility lets you conduct searches from the command line without writing code. If searchd returns a match, search queries the database and displays the rows in the match set. The search utility is useful for debugging your Sphinx configuration and performing impromptu searches.

In addition, the author of Sphinx, Andrew Aksyonoff, and other contributors provide APIs for PHP, Perl, C/C++, and other programming languages.

Search for body parts

Assume Body-Parts.com sells auto body parts — fenders, chrome, bumpers, etc. — for rare and collectible cars. As in the real world, a visitor to the Body Parts site is likely to search for parts by manufacturer (say, Porsche or a third party that makes equivalents), part number, make, model, year, condition (used, new, refurbished), and description, or some combination of those properties.

To build the Body Parts search feature, let's use MySQL V5.0 as the data store and the Sphinx search daemon to provide fast and accurate text search. MySQL V5.0 is a capable database, but its enhanced full-text search feature isn't especially rich. In fact, it's limited to MyISAM tables — a table format that doesn't support foreign keys and, thus, may be of limited use.

Listings 1 through 4 show the portion of the Body Parts schema relevant to this example. You see the Model (Listing 1), Assembly (Listing 2), Inventory (Listing 3), and Schematic (Listing 4) tables, respectively.

The Model table

The Model table shown in Listing 1 is simple: The label column enumerates the name of a model ("Corvette"); description acts as a consumer-friendly portrait of the auto ("Two-door roadster; first year of introduction"); and `begin_production` and `end_production` denote the years in which production of the version began and ended, respectively. Because the values in the aforementioned columns are not unique, a separate ID represents each quadruple (label, description, begin_production, end_production) and is a foreign key in other tables.

Listing 1. Body Parts Model table

```
CREATE TABLE Model (
  id int(10) unsigned NOT NULL auto_increment,
  label varchar(7) NOT NULL,
  description varchar(256) NOT NULL,
  begin_production int(4) NOT NULL,
  end_production int(4) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

Here's some sample data for the Model table:

```
INSERT INTO Model
  (`id`, `label`, `description`, `begin_production`, `end_production`)
VALUES
  (1,'X Sedan','Four-door performance sedan',1998,1999),
  (3,'X Sedan','Four door performance sedan, 1st model year',1995,1997),
  (4,'J Convertible','Two-door roadster, metal retracting roof',2002,2005),
  (5,'J Convertible','Two-door roadster',2000,2001),
  (7,'W Wagon','Four-door, all-wheel drive sport station wagon',2007,0);
```

The Assembly table

An *assembly* is a subsystem, such as the transmission or all the glass found on an auto. Owners consult an assembly drawing and associated parts list to find replacement parts. The Assembly table, shown in Listing 2, is also simple: It associates a unique ID with an assembly label and description.

Listing 2. The Assembly table

```
CREATE TABLE Assembly (
  id int(10) unsigned NOT NULL auto_increment,
  label varchar(7) NOT NULL,
  description varchar(128) NOT NULL,
  PRIMARY KEY (id)
) ENGINE=InnoDB;
```

To continue, here's some sample data for the Assembly table:

```
INSERT INTO Assembly
  (`id`, `label`, `description`)
VALUES
  (1,'5-00','Seats'),
  (2,'4-00','Electrical'),
  (3,'3-00','Glasses'),
  (4,'2-00','Frame'),
  (5,'1-00','Engine'),
  (7,'101-00','Accessories');
```

The Inventory table

The Inventory table is the canonical list of car parts. A part — such as a bolt or a bulb — might be in every car made and in multiple assemblies, but the part appears but once in the Inventory table. Each row in the Inventory table contains:

A unique 32-bit integer *serialno* used to identify the row.

An alphanumeric part number. (This part number is unique and could suffice as the primary key. However, because it can contain alphanumeric characters, it's inappropriate for use with Sphinx, which requires each record indexed to have a unique 32-bit integer key.)

A text description.

A price.

The specification for the Inventory table is shown in Listing 3.

Listing 3. The Inventory table

```
CREATE TABLE Inventory (
  id int(10) unsigned NOT NULL auto_increment,
  partno varchar(32) NOT NULL,
  description varchar(256) NOT NULL,
  price float unsigned NOT NULL default '0',
  PRIMARY KEY (id),
  UNIQUE KEY partno USING BTREE (partno)
) ENGINE=InnoDB;
```

A (partial) list of parts might look as follows:

```
INSERT INTO `Inventory`
  (`id`, `partno`, `description`, `price`)
VALUES
  (1,'WIN408','Portal window',423),
  (2,'ACC711','Jack kit',110),
  (3,'ACC43','Rear-view mirror',55),
  (4,'ACC5409','Cigarette lighter',20),
  (5,'WIN958','Windshield, front',500),
  (6,'765432','Bolt',0.1),
  (7,'ENG001','Entire engine',10000),
  (8,'ENG088','Cylinder head',55),
  (9,'ENG976','Large cylinder head',65);
```

The Schematic table

The Schematic table ties parts to assemblies and model versions. Hence, you'd use the Schematic table to find all parts that compose the engine in a 1979 J Class convertible. Each row in the Schematic table has a unique ID, a foreign key that refers to a row in the Inventory table, a foreign key that identifies the assembly, and another key to refer to a specific model and version in the Model table. The rows are shown in Listing 4.

Listing 4. The Schematic table

```
CREATE TABLE Schematic (
  id int(10) unsigned NOT NULL auto_increment,
  partno_id int(10) unsigned NOT NULL,
  assembly_id int(10) unsigned NOT NULL,
  model_id int(10) unsigned NOT NULL,
  PRIMARY KEY (id),
  KEY partno_index USING BTREE (partno_id),
  KEY assembly_index USING BTREE (assembly_id),
  KEY model_index USING BTREE (model_id),
  FOREIGN KEY (partno_id) REFERENCES Inventory(id),
  FOREIGN KEY (assembly_id) REFERENCES Assembly(id),
  FOREIGN KEY (model_id) REFERENCES Model(id)
) ENGINE=InnoDB;
```

To reinforce the purpose of the table, here's a small list of rows from Schematic:

```
INSERT INTO `Schematic`
  (`id`, `partno_id`, `assembly_id`, `model_id`)
VALUES
  (1,6,5,1),
  (2,8,5,1),
  (3,1,3,1),
  (4,5,3,1),
  (5,8,5,7),
  (6,6,5,7),
  (7,4,7,3),
  (8,9,5,3);
```

Searching with the tables

With these tables defined, a good many searches can easily be answered:

- Show all versions of a specific model
- List all assemblies required to assemble a particular model and version
- Show all parts in a particular assembly for a specific model and version

But a handful of searches are particularly costly:

- Find all occurrences of parts in any model and version with part numbers that begin with "WIN"
- Find those parts that have "lacquer" or "paint" in the description
- Find all the parts with "black leather" in the description
- Find all the 2002 J Series parts with "paint" in the description

Each of these searches could require voluminous `JOIN`s or expensive `LIKE` clauses, especially if the Inventory and Schematic tables are large. Further, complex text searches are simply beyond MySQL's

capabilities. To search vast quantities of text data, consider building and using a Sphinx index.

Integrate the Sphinx software

To apply Sphinx to a problem, you must define one or more sources and one or more indexes.

A *source* identifies the database to index, provides authentication information, and defines the query to use to construct each row. Optionally, a source can identify one or more columns as a filter, or what Sphinx calls a *group*. You use groups to filter results. For example, the word paint might yield 900 matches. If you're only interested in matches for a specific model car, you can further filter using the model group.

An *index* requires a source (that is, a set of rows) and defines how the data extracted from the source should be cataloged.

You define your source(s) and index(es) in the sphinx.conf file. The source for Body Parts is a MySQL database. Listing 5 shows part of the definition of the source named catalog — the snippet specifies which database to connect to and how to make a connection (host, socket, user, and password).

Listing 5. Settings to access the MySQL database

```
source catalog
{
    type                        = mysql

    sql_host                    = localhost
    sql_user                    = reaper
    sql_pass                    = s3cr3t
    sql_db                      = body_parts
    sql_sock                    =  /var/run/mysqld/mysqld.sock
    sql_port                    = 3306
```

Next, create a query to produce rows to be indexed. Typically, you create a `SELECT` statement, perhaps `JOIN`ing many tables together to yield a row. Here, though, there's a problem: A search for model and year must use the Assembly table, but the part number and part description can only be found in the Inventory table. To work, Sphinx must be able to tie a search result to a 32-bit integer primary key.

To get the data in the right form, create a *view*— a new construct in MySQL V5 that assembles columns from other tables into a single, composite virtual table. Using a view, all the data needed for every kind of search is in one place, even though the live data actually lives in other tables. Listing 6 shows the SQL to define the view named Catalog.

Listing 6. Catalog view assembles data into virtual table

```
CREATE OR REPLACE VIEW Catalog AS
SELECT
  Inventory.id,
  Inventory.partno,
  Inventory.description,
  Assembly.id AS assembly,
  Model.id AS model
FROM
  Assembly, Inventory, Model, Schematic
WHERE
  Schematic.partno_id=Inventory.id
  AND Schematic.model_id=Model.id
  AND Schematic.assembly_id=Assembly.id;
```

If you create a database named body_parts with the tables and data shown previously, your Catalog view should resemble this:

```
mysql> use body_parts;
Database changed
mysql> select * from Catalog;
+----+---------+---------------------+----------+-------+
| id | partno  | description         | assembly | model |
+----+---------+---------------------+----------+-------+
|  6 | 765432  | Bolt                |        5 |     1 |
```

```
|  8 | ENG088  | Cylinder head       |        5 |     1 |
|  1 | WIN408  | Portal window       |        3 |     1 |
|  5 | WIN958  | Windshield, front   |        3 |     1 |
|  4 | ACC5409 | Cigarette lighter   |        7 |     3 |
|  9 | ENG976  | Large cylinder head |        5 |     3 |
|  8 | ENG088  | Cylinder head       |        5 |     7 |
|  6 | 765432  | Bolt                |        5 |     7 |
+----+---------+---------------------+----------+-------+
8 rows in set (0.00 sec)
```

In the view, the field `id` points back to the part's entry in the Inventory table. The `partno` and `description` columns are the essential text to search, and the `assembly` and `model` columns serve as groups to further filter results. With the view in place, constructing the source query is a snap. Listing 7 shows the rest of the definition of the source named catalog.

Listing 7. A query to create rows to be indexed

```
    # indexer query
    # document_id MUST be the very first field
    # document_id MUST be positive (non-zero, non-negative)
    # document_id MUST fit into 32 bits
    # document_id MUST be unique
    sql_query                       = \
            SELECT \
                    id, partno, description, \
                    assembly, model \
            FROM \
                    Catalog;

    sql_group_column                = assembly
    sql_group_column                = model

    # document info query
    # ONLY used by search utility to display document information
    # MUST be able to fetch document info by its id, therefore
    # MUST contain '$id' macro
    #
    sql_query_info          = SELECT * FROM Inventory WHERE id=$id
}
```

The `sql_query` must include the primary key you want to use for subsequent lookups, and it must include all the fields you want to index and use as groups. The two `sql_group_column` entries declare that Assembly and Model can be used to filter results. And the search utility uses `sql_query_info` to find the records that match. In the query, `$id` is replaced with each primary key that searchd returns.

The last configuration step is to build an index. Listing 8 shows an index for the source catalog.

Listing 8. Describing one possible index for the source named catalog

```
index catalog
{
    source                  = catalog
    path                    = /var/data/sphinx/catalog
    morphology              = stem_en

    min_word_len            = 3
    min_prefix_len          = 0
    min_infix_len           = 3
}
```

Line 1 points to a named source in the sphinx.conf file. Line 2 defines where to store the index data; by convention, Sphinx indices are stored in /var/data/sphinx. Line 3 allows the index to use English morphology. And lines 5-7 tell the indexer to index only those words of three characters or more and to create an infix index of every substring of three characters or more. (For easy reference, Listing 9 shows the entire example sphinx.conf file for Body Parts.)

Listing 9. The example sphinx.conf for Body Parts

```
source catalog
{
    type                            = mysql

    sql_host                        = localhost
    sql_user                        = reaper
    sql_pass                        = s3cr3t
    sql_db                          = body_parts
    sql_sock                        =  /var/run/mysqld/mysqld.sock
    sql_port                        = 3306
```

```
    # indexer query
    # document_id MUST be the very first field
    # document_id MUST be positive (non-zero, non-negative)
    # document_id MUST fit into 32 bits
    # document_id MUST be unique

    sql_query                       = \
            SELECT \
                    id, partno, description, \
                    assembly, model \
            FROM \
                    Catalog;

    sql_group_column                = assembly
    sql_group_column                = model

    # document info query
    # ONLY used by search utility to display document information
    # MUST be able to fetch document info by its id, therefore
    # MUST contain '$id' macro
    #

    sql_query_info          = SELECT * FROM Inventory WHERE id=$id
}

index catalog
{
    source                  = catalog
    path                    = /var/data/sphinx/catalog
    morphology              = stem_en

    min_word_len            = 3
    min_prefix_len          = 0
    min_infix_len           = 3
}

searchd
{
        port                            = 3312
        log                                     = /var/log/searchd/searchd.log
        query_log                       = /var/log/searchd/query.log
        pid_file                        = /var/log/searchd/searchd.pid
}
```

The `searchd` section at bottom configures the searchd daemon itself. The entries in that section should be self-explanatory. The query.log is especially useful: It shows each search as it's run and displays results, such as the number of documents searched and the total number of matches.

Build and test the index

You are now ready to build the index for the Body Parts application. To do so:

1. Create the directory hierarchy /var/data/sphinx by typing: $ **sudo mkdir -p /var/data/sphinx**
2. Assuming that MySQL is running, run indexer to create the indices by using the code shown below.

   Listing 10. Create the indexes

   ```
   $ sudo /usr/local/bin/indexer --config /usr/local/etc/sphinx.conf --all
   Sphinx 0.9.7
   Copyright (c) 2001-2007, Andrew Aksyonoff

   using config file '/usr/local/etc/sphinx.conf'...
   indexing index 'catalog'...
   collected 8 docs, 0.0 MB
   sorted 0.0 Mhits, 82.8% done
   total 8 docs, 149 bytes
   total 0.010 sec, 14900.00 bytes/sec, 800.00 docs/sec
   ```

   Note: The `-all` argument rebuilds all the indexes listed in sphinx.conf. You can use a different argument to rebuild fewer if you don't need to rebuild every index.
3. You can now test the index with the search utility using the code shown below. (You don't need searchd running to use search.)

   Listing 11. Test the index with search

   ```
   $ /usr/local/bin/search --config /usr/local/etc/sphinx.conf ENG
   Sphinx 0.9.7
   Copyright (c) 2001-2007, Andrew Aksyonoff

   index 'catalog': query 'ENG ': returned 2 matches of 2 total in 0.000 sec

   displaying matches:
   1. document=8, weight=1, assembly=5, model=7
           id=8
           partno=ENG088
   ```

```
        description=Cylinder head
        price=55
2. document=9, weight=1, assembly=5, model=3
        id=9
        partno=ENG976
        description=Large cylinder head
        price=65

words:
1. 'eng': 2 documents, 2 hits

$ /usr/local/bin/search --config /usr/local/etc/sphinx.conf wind
Sphinx 0.9.7
Copyright (c) 2001-2007, Andrew Aksyonoff

index 'catalog': query 'wind ': returned 2 matches of 2 total in 0.000 sec

displaying matches:
1. document=1, weight=1, assembly=3, model=1
        id=1
        partno=WIN408
        description=Portal window
        price=423
2. document=5, weight=1, assembly=3, model=1
        id=5
        partno=WIN958
        description=Windshield, front
        price=500

words:
1. 'wind': 2 documents, 2 hits

$ /usr/local/bin/search \
--config /usr/local/etc/sphinx.conf --filter  model 3 ENG
Sphinx 0.9.7
Copyright (c) 2001-2007, Andrew Aksyonoff

index 'catalog': query 'ENG ': returned 1 matches of 1 total in 0.000 sec

displaying matches:
1. document=9, weight=1, assembly=5, model=3
        id=9
        partno=ENG976
        description=Large cylinder head
        price=65

words:
1. 'eng': 2 documents, 2 hits
```

The first command, `/usr/local/bin/search --config /usr/local/etc/sphinx.conf ENG`, found the two occurrences of `ENG` in part numbers. The second command, `/usr/local/bin/search --config /usr/local/etc/sphinx.conf wind`, found the substring `wind` in two part descriptions. And the third command restricted the results to those entries in which `model` is `3`.

Write the code

At long last, you can now write PHP code to call the Sphinx search engine. The Sphinx PHP API is small and easy to master. Listing 12 is a small PHP application to call searchd to extract the same results of the last command shown above ("find all parts with 'cylinder' in the name that belong to model 3").

Listing 12. Calling the Sphinx search engine from PHP

```
<?php
  include('sphinx-0.9.7/api/sphinxapi.php');

  $cl = new SphinxClient();
  $cl->SetServer( "localhost", 3312 );
  $cl->SetMatchMode( SPH_MATCH_ANY  );
  $cl->SetFilter( 'model', array( 3 ) );

  $result = $cl->Query( 'cylinder', 'catalog' );

  if ( $result === false ) {
      echo "Query failed: " . $cl->GetLastError() . ".\n";
  }
  else {
      if ( $cl->GetLastWarning() ) {
          echo "WARNING: " . $cl->GetLastWarning() . "
";
      }

      if ( ! empty($result["matches"]) ) {
          foreach ( $result["matches"] as $doc => $docinfo ) {
                echo "$doc\n";
          }

          print_r( $result );
```

```
    }
  }

  exit;
?>
```

To test the code, create the log directory for Sphinx, start searchd, then run the PHP application, as shown below.

Listing 13. The PHP application

```
$ sudo mkdir -p /var/log/searchd
$ sudo /usr/local/bin/searchd --config /usr/local/etc/sphinx.conf
$ php search.php
9
Array
(
    [fields] => Array
        (
            [0] => partno
            [1] => description
        )

    [attrs] => Array
        (
            [assembly] => 1
            [model] => 1
        )

    [matches] => Array
        (
            [9] => Array
                (
                    [weight] => 1
                    [attrs] => Array
                        (
                            [assembly] => 5
                            [model] => 3
                        )

                )

        )

    [total] => 1
    [total_found] => 1
    [time] => 0.000
    [words] => Array
        (
            [cylind] => Array
                (
                    [docs] => 2
                    [hits] => 2
                )

        )
)
```

The output is **9**: the correct primary key of the sole row that matches. If Sphinx makes a match, the associative array `$result` contains an element named `results`. Glance though the output of `print_r()` to see what else is returned.

One mention: `total_found` is the total number of matches found in the index, and `found` is the number of results returned. The two may differ because you can change how many matches are returned each time and which batch of matches to return, which is useful for paginating long lists of results. See the API call `SetLimits()`. One example of pagination is to call the search engine with: `$cl->SetLimits( ( $page - 1 ) * SPAN, SPAN )` to return the first, second, third (etc.) batch of `SPAN` matches, depending on which page is to be displayed.

Explore the mysteries of the Sphinx

Sphinx has many more features to take advantage of. I've barely scratched the surface here, but you now have a working real-world example to build on to expand your expertise.

Peruse the sample Sphinx configuration file, /usr/local/etc/sphinx.conf.dist, that ships with the distribution. The comments in that file explain what each Sphinx parameter can do; show how to create a distributed,

redundant configuration; and explain how to inherit base settings to avoid repetition in sources and indexes. The Sphinx README file is also a great source of information, including how to embed Sphinx directly into MySQL V5 — no daemons required.

Next time, search for better solutions than `echo()` and `print_r()` to debug PHP code.

Resources

Learn

Sphinx is an open source and free search engine designed to search text extremely quickly.

Check out Endeca or another large-scale commercial search product, or try Lucene.

PHP.net is the central resource for PHP developers.

Check out the "Recommended PHP reading list."

Browse all the PHP content on developerWorks.

Expand your PHP skills by checking out IBM developerWorks' PHP project resources.

To listen to interesting interviews and discussions for software developers, check out developerWorks podcasts.

Using a database with PHP? Check out the Zend Core for IBM, a seamless, out-of-the-box, easy-to-install PHP development and production environment that supports IBM DB2 V9.

Stay current with developerWorks' Technical events and webcasts.

Check out upcoming conferences, trade shows, webcasts, and other Events around the world that are of interest to IBM open source developers.

Visit the developerWorks Open source zone for extensive how-to information, tools, and project updates to help you develop with open source technologies and use them with IBM's products.

Watch and learn about IBM and open source technologies and product functions with the no-cost developerWorks On demand demos.

Get products and technologies

Looking for a database to use with your PHP applications? Download IBM DB2 Express-C 9, a no-charge version of DB2 Express V9 data server.

Innovate your next open source development project with IBM trial software, available for download or on DVD.

Download IBM product evaluation versions, and get your hands on application development tools and middleware products from DB2®, Lotus®, Rational®, Tivoli®, and WebSphere®.

Discuss

Participate in developerWorks blogs and get involved in the developerWorks community.

Dig deeper into Open source on developerWorks

- Overview
- New to Open source
- Projects
- Technical library (tutorials and more)
- Forums
- Events

Bluemix Developers Community
Get samples, articles, product docs, and community resources to help build, deploy, and manage your cloud apps.

developerWorks Labs
Experiment with new directions in software development.

DevOps Services
Software development in the cloud. Register today to create a project.

IBM evaluation software
Evaluate IBM software and solutions, and transform challenges into opportunities.

Participate in the developerWorks PHP Forum: Developing PHP applications with IBM Information Management products (DB2, IDS).