NAME

Dancer::Plugin::Auth::Extensible::Database − authenticate via a database

DESCRIPTION

This class is an authentication provider designed to authenticate users against a database, using Dancer::Plugin::Database to access a database.

Crypt::SaltedHash is used to handle hashed passwords securely; you wouldn't want to store plain text passwords now, would you? (If your answer to that is yes, please reconsider; you really don't want to do that, when it's so easy to do things right!)

See Dancer::Plugin::Database for how to configure a database connection appropriately; see the "CONFIGURATION" section below for how to configure this authentication provider with database details.

See Dancer::Plugin::Auth::Extensible for details on how to use the authentication framework, including how to pick a more useful authentication provider.

CONFIGURATION

This provider tries to use sensible defaults, so you may not need to provide much configuration if your database tables look similar to those in the "SUGGESTED SCHEMA" section below.

The most basic configuration, assuming defaults for all options, and defining a single authentication realm named 'users':

```
plugins:
    Auth::Extensible:
        realms:
            users:
                provider: 'Database'
```

You would still need to have provided suitable database connection details to Dancer::Plugin::Database, of course; see the docs for that plugin for full details, but it could be as simple as, e.g.:

```
plugins:
    Auth::Extensible:
        realms:
            users:
                provider: 'Database'
    Database:
        driver: 'SQLite'
        database: 'test.sqlite'
```

A full example showing all options:

```
plugins:
    Auth::Extensible:
        realms:
            users:
                provider: 'Database'
                # optionally set DB connection name to use (see named
                # connections in Dancer::Plugin::Database docs)
                db_connection_name: 'foo'

                # Optionally disable roles support, if you only want to check
                # for successful logins but don't need to use role-based acce
                disable_roles: 1

                # optionally specify names of tables if they're not the defau
                # (defaults are 'users', 'roles' and 'user_roles')
                users_table: 'users'
                roles_table: 'roles'
                user_roles_table: 'user_roles'

                # optionally set the column names (see the SUGGESTED SCHEMA
```

```
# section below for the default names; if you use them, they'
# Just Work)
users_id_column: 'id'
users_username_column: 'username'
users_password_column: 'password'
roles_id_column: 'id'
roles_role_column: 'role'
user_roles_user_id_column: 'user_id'
user_roles_role_id_column: 'roles_id'
```

See the main Dancer::Plugin::Auth::Extensible documentation for how to configure multiple authentication realms.

SUGGESTED SCHEMA

If you use a schema similar to the examples provided here, you should need minimal configuration to get this authentication provider to work for you.

The examples given here should be MySQL-compatible; minimal changes should be required to use them with other database engines.

users table

You'll need a table to store user accounts in, of course. A suggestion is something like:

```
CREATE TABLE users (
    id       INTEGER     AUTO_INCREMENT PRIMARY KEY,
    username VARCHAR(32) NOT NULL       UNIQUE KEY,
    password VARCHAR(40) NOT NULL
);
```

You will quite likely want other fields to store e.g. the user's name, email address, etc; all columns from the users table will be returned by the `logged_in_user` keyword for your convenience.

roles table

You'll need a table to store a list of available roles in (unless you're not using roles − in which case, disable role support (see the ''CONFIGURATION'' section).

```
CREATE TABLE roles (
    id    INTEGER     AUTO_INCREMENT PRIMARY KEY,
    role  VARCHAR(32) NOT NULL
);
```

user_roles table

Finally, (unless you've disabled role support) you'll need a table to store user <−> role mappings (i.e. one row for every role a user has; so adding extra roles to a user consists of adding a new role to this table). It's entirely up to you whether you use an ''id'' column in this table; you probably shouldn't need it.

```
CREATE TABLE user_roles (
    user_id  INTEGER  NOT NULL,
    role_id  INTEGER  NOT NULL,
    UNIQUE KEY user_role (user_id, role_id)
);
```

If you're using InnoDB tables rather than the default MyISAM, you could add a foreign key constraint for better data integrity; see the MySQL documentation for details, but a table definition using foreign keys could look like:

```
CREATE TABLE user_roles (
    user_id  INTEGER, FOREIGN KEY (user_id) REFERENCES users (id),
    role_id  INTEGER, FOREIGN_KEY (role_id) REFERENCES roles (id),
    UNIQUE KEY user_role (user_id, role_id)
) ENGINE=InnoDB;
```