Doctrine spatial extension

Release 3.0.0

Alexandre Tranchant

Jan 13, 2022

CONTENTS

Doctrine spatial extension provides spatial types and spatial functions for doctrine. It allows you to manage spatial entity and to store them into your database server.

Currently, doctrine spatial extension provides two dimension general geometric and geographic spatial types, two-dimension points, linestrings, polygon and two-dimension multi-points, multi-linestrings, multi-polygons. Doctrine spatial is only compatible with MySql and PostgreSql. For better security and better resilience of your spatial data, we recommend that you favor the PostgreSql database server because of the shortcomings and vulnerabilities of MySql.

This project was initially created by Derek J. Lambert in 2015. In March 2020, Alexandre Tranchant forked the originally project because of unactivity for two years. Feel free to *contribute*. Any help is welcomed:

- to implement third and fourth dimension in spatial data,
- to implement new spatial function,
- to complete documentation and fix typos, *(I'm not fluent in english)*
- to implement new abstracted platforms like Microsoft Sql Server.

CHAPTER

ONE

CONTENTS

1.1 Installation

1.1.1 Installation via composer

Add the *longitude-one/doctrine-spatial* package in your composer.json.

```
$ composer require longitude-one/doctrine-spatial
```

Or you can edit directly *composer.json* file by adding this line on your requirements:

```
"require": {
    ...
    "longitude-one/doctrine-spatial": "^3.0"
```

Be careful, the version 3.0.0 will only be available during summer 2021.

1.1.2 Installation without composer

If you're not using composer, you can `download the library`. Then copy and paste the lib directory where you store all libraries of your application.

1.1.3 Installation to contribute

If you want to contribute, do not hesitate. Any help is welcomed. The simplest way is too fork the Github project, then to locally clone your forked project. The *contribution* page explains how to proceed and how to test your updates.

1.2 Configuration

1.2.1 Configuration for applications using Symfony framework

To configure Doctrine spatial extension on your Symfony application, you only need to edit your `config/doctrine.yaml` file. Two steps are sufficient. First step will help you to declare spatial types on DQL. The second step will help you to declare a spatial function.

1.2.1.1 Declare your geometric types

```
doctrine:
    dbal:
        types:
            geometry:   LongitudeOne\Spatial\DBAL\Types\GeometryType
            point:      LongitudeOne\Spatial\DBAL\Types\Geometry\PointType
            polygon:    LongitudeOne\Spatial\DBAL\Types\Geometry\PolygonType
            linestring: LongitudeOne\Spatial\DBAL\Types\Geometry\LineStringType
```

Now, you can *create an entity* with a `geometry`, `point`, `polygon` and a `linestring` type.

Here is a complete example of all available types. The names of doctrine types are not hardcoded. So if you only want to use the geometric type, feel free to remove the `geometric_` prefixes

```
doctrine:
    dbal:
        types:
            geography:            LongitudeOne\Spatial\DBAL\Types\GeographyType
            geography_linestring: LongitudeOne\Spatial\DBAL\Types\Geography\
↪LineStringType
            geography_point:      LongitudeOne\Spatial\DBAL\Types\Geography\PointType
            geography_polygon:    LongitudeOne\Spatial\DBAL\Types\Geography\PolygonType

            geometry:            LongitudeOne\Spatial\DBAL\Types\GeometryType
            geometry_linestring: LongitudeOne\Spatial\DBAL\Types\Geometry\LineStringType
            geometry_point:      LongitudeOne\Spatial\DBAL\Types\Geometry\PointType
            geometry_polygon:    LongitudeOne\Spatial\DBAL\Types\Geometry\PolygonType
            geometry_multilinestring: LongitudeOne\Spatial\DBAL\Types\Geometry\
↪MultiLineStringType
            geometry_multipoint:      LongitudeOne\Spatial\DBAL\Types\Geometry\
↪MultiPointType
            geometry_multipolygon:    LongitudeOne\Spatial\DBAL\Types\Geometry\
↪MultiPolygonType
```

I try to maintain this documentation up-to-date. In any case, the DBAL/Types directory contains all geometric and all geographic available types.

Any help is welcomed to implement the other spatial types declared in the Open Geospatial Consortium standard and in the ISO/IEC 13249-3:2016 like `Curve` or `PolyhedSurface`.

1.2.1.2 Declare a new function

```
orm:
    dql:
        numeric_functions:
            #Declare functions returning a numeric value
            #A good practice is to prefix functions with ST when they are issue from the␣
↪Standard directory
            st_area: LongitudeOne\Spatial\ORM\Query\AST\Functions\Standard\StArea
        string_functions:
            #Declare functions returning a string
            st_envelope: LongitudeOne\Spatial\ORM\Query\AST\Functions\Standard\STEnvelope
```

(continues on next page)

(continued from previous page)

```
            #Prefix functions with SP when they are not issue from the Standard␣
↪directory is a good practice
            sp_asgeojson: LongitudeOne\Spatial\ORM\Query\AST\Functions\Postgresql\
↪SpAsGeoJson
            #You can use the DQL function name you want and then use it in your DQL
            myDQLFunctionAlias: LongitudeOne\Spatial\ORM\Query\AST\Functions\Standard\
↪StCentroid
            #SELECT myDQLFunctionAlias(POLYGON(...
```

Add only the functions you want to use. The list of available function can be found in these sections:

1. list of *Functions described in OGC Standards or in ISO/IEC 13249-3:2016* declared in the Open Geospatial Consortium standard,
2. list of *Specific functions of the PostgreSql database server* which are not already declared in the OGC Standard,
3. list of *Specific functions of the MySql database server* which are not already declared in the OGC Standard,

Nota: By default, function declared by the Open Geospatial Consortium in the standards of SQL Options are prefixed by `ST_`, other functions should not be declared with this prefix. We suggest to use the `SP_` prefix (specific).

1.2.2 Configuration for other application

1.2.2.1 Declare your geometric types

Doctrine allows you to create new mapping types. We used this functionnality to create spatial types in this extension. You only need to let Doctrine know which type you want to use. Two lines are sufficient to do it. The first line calls the `Type` class. The second line, declare a type. In the below example, we declare a geometric `point` type.

```
<?php
// in your bootstrapping code

// ...

use Doctrine\DBAL\Types\Type;

// ...

// Register types provided by the doctrine2 spatial extension
Type::addType('point', 'LongitudeOne\Spatial\DBAL\Types\Geometry\PointType');
```

1.2.2.2 Declare a new function

You can register functions of the doctrine spatial extension adding them to the ORM configuration:

```
<?php

// in your bootstrapping code

// ...

use Doctrine\ORM\Configuration\Doctrine\ORM\Configuration;
```

(continues on next page)

(continued from previous page)

```
// ...

$config = new Configuration();
// This is an example to declare a standard spatial function which is returning a string
$config->addCustomStringFunction('ST_Envelope', 'LongitudeOne\Spatial\ORM\Query\AST\
↪Functions\Standard\StEnvelope');
// This is another example to declare a standard spatial function which is returning a␣
↪numeric
$config->addCustomNumericFunction('ST_Area', 'LongitudeOne\Spatial\ORM\Query\AST\
↪Functions\Standard\StArea');
// This is another example to declare a Postgresql specific function which is returning␣
↪a string
$config->addCustomNumericFunction('SP_GeoJson', 'LongitudeOne\Spatial\ORM\Query\AST\
↪Functions\PostgreSql\SpGeoJson');
```

1.3 Create spatial entities

It is a good practice to use the most adapted column to store you geometric or geographic data. If your entity have only to store points, do not use a "geometry" type, but a "point" type. Use a geometry column only if your entity can store different types (points and lines as example)

1.3.1 Example1: Entity with a spatial point

Below, you will find an example to declare an entity with a `point`. Before you need to declare the point type as described in the *configuration section*.

```
<?php

// We declare the class of the type we want to use.
use LongitudeOne\Spatial\PHP\Types\Geometry\Point;
// We declare the Mapping as usual
use Doctrine\ORM\Mapping as ORM;

/**
 * Point entity example.
 *
 * As you can see we do not change anything in Entity and Table annotations. Feel free␣
↪to use them as usual.
 *
 * @ORM\Entity
 * @ORM\Table
 */
class PointEntity
{
    /**
     * @var int The usual Doctrine Identifier
     *
     *  As you can see we do not change anything in Entity and Table annotations. Feel␣
↪free to use it as usual.
```

(continues on next page)

(continued from previous page)

```
     *
     * @ORM\Id
     * @ORM\GeneratedValue(strategy="AUTO")
     * @ORM\Column(type="integer")
     */
    protected $id;

    /**
     * @var Point
     *
     * As you can see we declare a point property of type point.
     * point shall be declared in the doctrine.yaml as a custom type.
     * Feel free to use options as usual. As example, I declared that point is not␣
↪nullable. But you can
     * set it to nullable=true if you want.
     *
     * @ORM\Column(type="point", nullable=false)
     */
    protected $point;

    /**
     * Get id.
     * This is the usual Id getter.
     *
     * @return int
     */
    public function getId()
    {
        return $this->id;
    }

    /**
     * Get point.
     * This is a standard getter.
     *
     * @return Point
     */
    public function getPoint(): Point
    {
        return $this->point;
    }

    /**
     * Set point.
     * This is a fluent setter.
     *
     * @param Point $point point to set
     *
     * @return self
     */
    public function setPoint(Point $point): self
    {
```

(continues on next page)

(continued from previous page)

```
        $this->point = $point;

        return $this;
    }
}
```

1.3.2 Seven examples with each geometric spatial types

The Fixtures directory creates some spatial entities for our tests. Inside this directory, you will find a lot of entities which are implementing geometric properties:

- Entity with a geometric type, `download`
- Entity with a geometric linestring type, `download`
- Entity with a geometric multilinestring type, `download`
- Entity with a geometric multipoint type, `download`
- Entity with a geometric multipolygon type, `download`
- Entity with a geometric point type, `download`
- Entity with a geometric polygon type. `download`

1.3.3 Four examples with each geographic spatial types

The Fixtures directory creates some spatial entities for our tests. Inside this directory, you will find a lot of entities which are implementing geographic properties:

- Entity with a geographic type, `download`
- Entity with a geographic linestring type, `download`
- Entity with a geographic point type, `download`
- Entity with a geographic polygon type, `download`

1.4 Repository

When your spatial entity is created, you can add new methods to your repositories. This section will explain you how to add new methods to a standard repository.

In this example, we assume that a building entity was already created. The building entity owns a spatial property to store polygon. We assume that the entity is named `building` and that the spatial property is name `plan` which is a `polygon` type.

```
<?php

namespace App\Repository;

use App\Entity\Building; // This is our spatial entity
use Doctrine\Bundle\DoctrineBundle\Repository\ServiceEntityRepository;
use Doctrine\Persistence\ManagerRegistry;
```

(continues on next page)

(continued from previous page)

```php
/**
 * Building repository.
 *
 * These methods inherits from ServiceEntityRepository
 *
 * @method Building|null find($id, $lockMode = null, $lockVersion = null)
 * @method Building|null findOneBy(array $criteria, array $orderBy = null)
 * @method Building[]    findAll()
 * @method Building[]    findBy(array $criteria, array $orderBy = null, $limit = null,
↪$offset = null)
 */
class BuildingRepository extends ServiceEntityRepository
{
    /**
     * BuildingRepository constructor.
     *
     * The repository constructor of a spatial entity is strictly identic to other␣
↪repositories.
     *
     * @param ManagerRegistry $registry injected by dependency injection
     */
    public function __construct(ManagerRegistry $registry)
    {
        parent::__construct($registry, Building::class);
    }

    // ...

    /**
     * Find building that have an area between min and max .
     *
     * @param float $min the minimum accepted area
     * @param float $max the maximum accepted area
     *
     * @return Building[]
     */
    public function findAreaBetween(float $min, float $max): array
    {
        //The query builder is normally retrieved
        $queryBuilder = $this->createQueryBuilder('b');

        //We assume that the ST_AREA has been declared in configuration
        return $queryBuilder->where('ST_AREA(b.plan) BETWEEN :min AND :max')
            ->setParameter('min', $min, 'float')
            ->setParameter('max', $max, 'float')
            ->getQuery()
            ->getResult()
        ;
    }
```

1.5 Glossary

1.5.1 Types

1.5.1.1 Types described in OGC Standards or in ISO/IEC 13249-3:2016

The ISO/IEC 13249-3 International Standard defines multimedia and application specific types and their associated routines using the user-defined features in ISO/IEC 9075. The third part of ISO/IEC 13249 defines spatial user-defined types and their associated routines.

In doctrine spatial extensions, some of all normalized spatial user-defined types are implemented.

This section lists them.

Spatial types	COORDINATES	Implemented	MySql	PostgreSql
Geometric	X, Y	YES	YES	YES
Point	X, Y	YES	YES	YES
LineString	X, Y	YES	YES	YES
Polygon	X, Y	YES	YES	YES
MultiPoint	X, Y	YES	YES	YES
MultiLineString	X, Y	YES	YES	YES
MultiPolygon	X, Y	YES	YES	YES
GeomCollection	X, Y	NO		
Curve	X, Y	NO		
Surface	X, Y	NO		
PolyHedralSurface	X, Y	NO		
GeometricZ	X, Y, Z	NO		
PointZ	X, Y, Z	NO		
LineStringZ	X, Y, Z	NO		
PolygonZ	X, Y, Z	NO		
MultiPointZ	X, Y, Z	NO		
MultiLineStringZ	X, Y, Z	NO		
MultiPolygonZ	X, Y, Z	NO		
GeomCollectionZ	X, Y, Z	NO		
CurveZ	X, Y, Z	NO		
SurfaceZ	X, Y, Z	NO		
PolyHedralSurfaceZ	X, Y, Z	NO		
GeometricM	X, Y, M	NO		
PointM	X, Y, M	NO		
LineStringM	X, Y, M	NO		
PolygonM	X, Y, M	NO		
MultiPointM	X, Y, M	NO		
MultiLineStringM	X, Y, M	NO		
MultiPolygonM	X, Y, M	NO		
GeomCollectionM	X, Y, M	NO		
CurveM	X, Y, M	NO		
SurfaceM	X, Y, M	NO		
PolyHedralSurfaceM	X, Y, M	NO		
GeometricZM	X, Y, Z, M	NO		
PointZM	X, Y, Z, M	NO		
LineStringZM	X, Y, Z, M	NO		

continues on next page

Table 1 – continued from previous page

Spatial types	COORDINATES	Implemented	MySql	PostgreSql
PolygonZM	X, Y, Z, M	NO		
MultiPointZM	X, Y, Z, M	NO		
MultiLineStringZM	X, Y, Z, M	NO		
MultiPolygonZM	X, Y, Z, M	NO		
GeomCollectionZM	X, Y, Z, M	NO		
CurveZM	X, Y, Z, M	NO		
SurfaceZM	X, Y, Z, M	NO		
PolyHedralSurfaceZM	X, Y, Z, M	NO		

1.5.2 Functions

1.5.2.1 Functions described in OGC Standards or in ISO/IEC 13249-3:2016

The ISO/IEC 13249-3 International Standard defines multimedia and application specific types and their associated routines using the user-defined features in ISO/IEC 9075. The third part of ISO/IEC 13249 defines spatial user-defined types and their associated routines.

Associated routines of this document are considered as the "Standard functions" for this doctrine spatial extension. I try to maintain this documentation up-to-date. In any case, you will find under the Functions/Standards directory a set of classes. Each class implement the spatial function of the same name.

The below table shows the defined functions:

Spatial functions	Implemented	Type	MySql	PostgreSql
ST_Area	YES	Numeric	YES	YES
ST_AsBinary	YES	String	YES	YES
ST_Boundary	YES	String	YES	YES
ST_Buffer	YES	Numeric	NO*	YES
ST_Centroid	YES	String	YES	YES
ST_Contains	YES	Numeric	YES	YES
ST_ConvexHull	YES	String	NO	YES
ST_Crosses	YES	Numeric	YES	YES
ST_Difference	YES	String	YES	YES
ST_Dimension	YES	Numeric	YES	YES
ST_Disjoint	YES	Numeric	YES	YES
ST_Distance	YES	Numeric	NO*	YES
ST_Equals	YES	Numeric	YES	YES
ST_Intersects	YES	Numeric	YES	YES
ST_Intersection	YES	String	YES	YES
ST_IsClosed	YES	Numeric	YES	YES
ST_IsEmpty	YES	Numeric	YES	YES
ST_IsRing	YES	Numeric	NO	YES
ST_IsSimple	YES	Numeric	YES	YES
ST_EndPoint	YES	String	YES	YES
ST_Envelope	YES	String	YES	YES
ST_ExteriorRing	YES	String	YES	YES
ST_GeometryN	YES	String	YES	YES
ST_GeometryN	YES	String	YES	YES
ST_EndPoint	YES	String	YES	YES

continues on next page

Table 2 – continued from previous page

Spatial functions	Implemented	Type	MySql	PostgreSql
ST_GeometryType	YES	Numeric	NO*	YES
ST_GeomFromWkb	YES	String	YES	YES
ST_GeomFromText	YES	String	YES	YES
ST_InteriorRingN	YES	String	YES	YES
ST_Length	YES	Numeric	YES	YES
ST_LineStringFromWkb	YES	String	YES	YES
ST_MPointFromWkb	YES	String	YES	YES
ST_MLineFromWkb	YES	String	YES	YES
ST_MPolyFromWkb	YES	String	YES	YES
ST_NumInteriorRing	YES	String	YES	YES
ST_NumGeometries	YES	String	YES	YES
ST_NumPoints	YES	String	YES	YES
ST_Overlaps	YES	String	YES	YES
ST_Perimeter	YES	String	YES	YES
ST_Point	YES	String	YES	YES
ST_PointFromWkb	YES	String	YES	YES
ST_PointN	YES	String	YES	YES
ST_PointOnSurface	YES	String	NO	YES
ST_PolyFromWkb	YES	String	YES	YES
ST_Relate	YES	String	YES	YES
ST_SetSRID	YES	Numeric	YES	YES
ST_StartPoint	YES	Numeric	YES	YES
ST_SymDifference	YES	String	YES	YES
ST_Touches	YES	Numeric	YES	YES
ST_Union	YES	String	YES	YES
ST_Within	YES	Numeric	YES	YES
ST_X	YES	Numeric	YES	YES
ST_Y	YES	Numeric	YES	YES

1.5.2.2 Specific functions of the PostgreSql database server

If your application can be used with another database server than PostgreSql, you should avoid to use these functions. It's a good practice to name function with the SP prefix, but do not forget that you can name all functions as you want when you declare it into your configuration files or in your bootstrap.

Specific PostgreSQL Spatial functions	Implemented	Type
Sp_AsGeoJson	YES	String
Sp_Azimuth	YES	String
Sp_ClosestPoint	YES	String
Sp_Collect	YES	String
Sp_ContainsProperly	YES	Numeric
Sp_CoveredBy	YES	Numeric
Sp_Covers	YES	Numeric
Sp_Distance_Sphere	YES	Numeric
Sp_DWithin	YES	Numeric
Sp_Expand	YES	Numeric
Sp_GeogFromText	YES	String
Sp_GeographyFromText	YES	String

continues on next page

Table 3 – continued from previous page

Specific PostgreSQL Spatial functions	Implemented	Type
Sp_GeomFromEwkt	YES	Numeric
Sp_GeometryType	YES	Numeric
Sp_LineCrossingDirection	YES	Numeric
Sp_LineSubstring	YES	Numeric
Sp_LineLocatePoint	YES	Numeric
Sp_LineInterpolatePoint	YES	String
Sp_MakeEnvelope	YES	String
Sp_MakeBox2D	YES	String
Sp_MakeLine	YES	String
Sp_MakePoint	YES	String
Sp_NPoints	YES	Numeric
Sp_Scale	YES	Numeric
Sp_Simplify	YES	Numeric
Sp_Split	YES	Numeric
Sp_SnapToGrid	YES	String
Sp_Summary	YES	String
Sp_Transform	YES	Numeric
Sp_Translate	YES	Numeric

1.5.2.3 Specific functions of the MySql database server

If your application can be used with another database server than MySql, you should avoid to use these functions.

Specific MySQL Spatial functions	Implemented	Type
Sp_Distance	YES	Numeric
Sp_Buffer	YES	Numeric
Sp_BufferStrategy	YES	Numeric
Sp_GeometryType	YES	Numeric
Sp_LineString	YES	Numeric
Sp_MBRContains	YES	Numeric
Sp_MBRDisjoint	YES	Numeric
Sp_MBREquals	YES	Numeric
Sp_MBRDisjoint	YES	Numeric
Sp_MBRIntersects	YES	Numeric
Sp_MBROverlaps	YES	Numeric
Sp_MBRTouches	YES	Numeric
Sp_MBRWithin	YES	Numeric
Sp_Point	YES	Numeric

Nota: Since MySql 5.7, a lot of functions are deprecated. These functions have been removed from doctrine spatial extensions, because they are replaced by their new names. As example, the GeomFromText function does no more exist. It has been replaced by the Standard function ST_GeomFromText since MySql 5.7. So if you was using GeomFromText, removed it and use the standard function declared in the StGeomFromText class.

1.6 Contributing

1.6.1 Documentation

This documentation is done with sphinx. All documentation are stored in the `docs` directory. To contribute to this documentation (and fix the lot of typo), you could install docker and start the docker service named `spatial_doc`. It's included in the repository. It will self-compile the documentation. After you start this service, you can now access the documentation at http://localhost:8100. If you try changing any `rst` file, after some seconds the browser auto refresh to show the updated documentation. Following the Sphinx documentation site you can now document this project using Sphinx.

1. Edit files in the `docs` directory,
2. Verify that documentation is improved,
3. Commit your contribution with an explicit message,
4. Push your commit and create a pull request to the longitude-one/doctrine-spatial project.

1.6.2 Source code

1.6.2.1 How to create a new function?

It's pretty easy to create a new function. A few lines code are sufficient.

1.6.2.1.1 Where to store your class?

If your function is described in the OGC Standards or in the ISO/IEC 13249-3, the class implementing the function **shall** be create in the lib/LongitudeOne/Spatial/ORM/Query/AST/Functions/Standard directory.

If your spatial function is not described in the OGC Standards nor in the ISO, your class should be prefixed by Sp (specific). If your class is specific to MySql, you shall create it in the lib/LongitudeOne/Spatial/ORM/Query/AST/Functions/MySql directory. If your class is specific to PostgreSQL, you shall create it in the lib/LongitudeOne/Spatial/ORM/Query/AST/Functions/PostgreSql directory. If your class is not described in the OGC Standards nor in the ISO norm, but exists in MySQL and in PostgreSQL, accepts the same number of arguments and returns the same results (which is rarely the case), then you shall create it in the lib/LongitudeOne/Spatial/ORM/Query/AST/Functions/Common directory.

1.6.2.1.2 Which name for your function?

Create a new class. It's name shall be the same than the function name in camel case prefixed with `St` or `Sp`. The standards are alive, they can be updated at any time. Regularly, new spatial function are defined by consortium. So, to avoid that a new standardized function as the same name from an existing function, the `St` prefix is reserved to already standardized function.

If your function is described in the OGC Standards or in the ISO/IEC 13249-3, the prefix shall be `St` else your class shall be prefixed with `Sp`. As example, if you want to create the spatial `ST_Z` function, your class shall be named `StZ` in the Standard directory. If you want to create the ST_Polygonize PostgreSql function which is not referenced in the OGC nor in ISO, then you shall name your class `SpPolygonize` and store them in the PostgreSql directory.

1.6.2.1.3 Which method to implements?

Now you know where to create your class, it should extends `AbstractSpatialDQLFunction` and you have to implement four functions:

1. `getFunctionName()` shall return the SQL function name,
2. `getMaxParameter()` shall return the maximum number of arguments accepted by the function,
3. `getMinParameter()` shall return the minimum number of arguments accepted by the function,
4. `getPlatforms()` shall return an array of each platform accepting this function.

As example, if the new spatial function exists in PostgreSQL and in MySQL, `getPlatforms()` should be like this:

```
<?php

// ...

/**
 * Get the platforms accepted.
 *
 * @return string[] a non-empty array of accepted platforms
 */
protected function getPlatforms(): array
{
    return ['postgresql', 'mysql'];
}
```

Do not hesitate to copy and paste the implementing code of an existing spatial function.

If your function is more specific and need to be parse, you can overload the parse method. The PostgreSQL SnapToGrid can be used as example.

All done! Your function is ready to used, but, please, read the next section to implement tests.

Don't forget to check your code respect our standard of quality:

```
docker exec spatial-php7 composer check-quality-code
```

1.6.2.2 How to test your new function?

Please, create a functional test in the same way. You have a lot of example in the functions test directory.

1.6.2.2.1 Setup

Here is an example of setup, each line is commented to help you to understand how to setup your test.

```
<?php

use LongitudeOne\Spatial\Exception\InvalidValueException;
use LongitudeOne\Spatial\Exception\UnsupportedPlatformException;
use LongitudeOne\Spatial\Tests\Helper\PointHelperTrait;
use LongitudeOne\Spatial\Tests\OrmTestCase;
use Doctrine\DBAL\Exception;
```

(continues on next page)

(continued from previous page)

```php
use Doctrine\ORM\ORMException;

/**
 * Foo DQL functions tests.
 * These tests verify their implementation in doctrine spatial.
 *
 * @author  Alexandre Tranchant <alexandre.tranchant@gmail.com>
 * @license https://alexandre-tranchant.mit-license.org MIT
 *
 * Please preserve the three above annotation.
 *
 * Group is used to exclude some tests on some environment.
 * Internal is to avoid the use of the test outer of this library
 * CoversDefaultClass is to avoid that your test covers other class than your new class
 *
 * @group dql
 *
 * @internal
 * @coversDefaultClass
 */
class SpFooTest extends OrmTestCase
{
    // To help you to create some geometry, I created some Trait.
    // use it to be able to call some methods which will store geometry into your␣
↪database
    // In this example, we use a trait that will create some points.
    use PointHelperTrait;

    /**
     * Setup the function type test.
     */
    protected function setUp(): void
    {
        //If you create point entity in your test, you shall add the line above or the␣
↪**next** test will failed
        $this->usesEntity(self::POINT_ENTITY);
        //If the method exists in mysql, You shall test it. Comment this line if␣
↪function does not exists on MySQL
        $this->supportsPlatform('mysql');
        //If the method exists in postgresql, You shall test it. Comment this line if␣
↪function does not exists on PostgreSql
        $this->supportsPlatform('postgresql');

        parent::setUp();
    }

    /**
     * Test a DQL containing function to test in the select.
     */
    public function testSelectSpBuffer()
    {
        //The above protected method come from the point helper trait.
```

(continues on next page)

(continued from previous page)

```
        //It creates a point at origin (0 0) and persist it in database
        $point0 = $this->persistPointOrigin();

        //We create a query using your new DQL function SpFoo
        $query = $this->getEntityManager()->createQuery(
            'SELECT p, ST_AsText(SpFoo(p.point, :p) FROM LongitudeOne\Spatial\Tests\
↪Fixtures\PointEntity p'
        );
        //Optionnaly, you can use parameter
        $query->setParameter('p', 'bar', 'string');
        //We retrieve the result
        $result = $query->getResult();

        //Now we test the result
        static::assertCount(1, $result);
        static::assertEquals($point0, $result[0][0]);
        static::assertSame('POLYGON((-4 -4,4 -4,4 4,-4 4,-4 -4))', $result[0][1]);
    }
```

Now, open the OrmTestCase.php file] and declare your function in one of this three methods:

- `addStandardFunctions`
- `addMySqlFunctions`
- `addPostgreSqlFunctions`

You can launch the test. This *document* helps you how to config your dev environment. Please do not forgot to update documentation by adding your function in one of these three tables:

- *Functions described in OGC Standards or in ISO/IEC 13249-3:2016*
- *Specific functions of the MySql database server*
- *Specific functions of the PostgreSql database server*

1.6.3 Quality of your code

Quality of code is auto-verified by php-cs-fixer, php code sniffer and php mess detector.

Before a commit, launch the quality script:

```
docker spatial-php8 composer check-quality-code
```

You can launch PHPCS-FIXER to fix errors with:

```
docker spatial-php8 composer phpcsfixer
```

You can launch PHP Code Sniffer only with: .. code-block:: bash

    docker spatial-php8 composer phpcs

You can launch PHP Mess Detector only with:

```
docker spatial-php8 composer phpmd
```

1.7 Test environment

If you want to contribute to this library, you're welcome. This section will help you to prepare your development environment.

1.7.1 How to prepare environment?

Doctrine library is available for MySQL and PostGreSQL.

1. [Install docker](https://docs.docker.com/engine/install/),
2. Go to the docker directory and start docker

```
cd <project_directory>
cd docker
docker-compose up -d
cd ..
```

Done! Your environment is ready with five services:

1. A MySQL5.7 service, you can connect to database with mysql://main@main:127.0.0.1:3357/main/
2. A MySQL8.0 service, you can connect to database with mysql://main@main:127.0.0.1:3380/main
3. A PostGreSQL service, you can connect to database with mysql://main@main:127.0.0.1:5432/main
4. A PHP7.4 service, you can test it via: docker exec spatial-php7 php -v
5. A PHP8.0 service, you can test it via: docker exec spatial-php8 php -v

Composer is installed on spatial-php7 and spatial-php8.

```
docker exec spatial-php7 composer -v
docker exec spatial-php8 composer -v
```

1.7.2 How to start test?

Copy docker/phpunit.*.xml to the project directory

```
cp docker/phpunit.*.xml .
```

Then, you can launch the test on php7, then php8:

```
docker exec spatial-php7 composer test-mysql5.7
docker exec spatial-php7 composer test-mysql8.0
docker exec spatial-php7 composer test-pgsql
docker exec spatial-php8 composer test-mysql5.7
docker exec spatial-php8 composer test-mysql8.0
docker exec spatial-php8 composer test-pgsql
```

After any update, before any commit, simply check your code with this composer command:

```
docker exec spatial-php7 composer check-quality-code
```