Logical Authorization Bundle Documentation

Release

Kristofer Tengström

July 14, 2019

Contents

CHAPTER 1

Overview

1.1 Requirements

This bundle requires Symfony 4.1 or higher.

1.2 Installation

Main bundle:

```
composer require ordermind/logical-authorization-bundle
```

Support for Doctrine ORM:

```
composer require ordermind/logical-authorization-doctrine-orm-bundle
```

Support for Doctrine MongoDB:

```
composer require ordermind/logical-authorization-doctrine-mongo-bundle
```

1.3 License

OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

1.4 Contribution Guidelines

1. Make sure that your code is formatted according to the Symfony coding standards at https://symfony.com/doc/current/contributing/code/standards.html
2. All pull requests must include unit tests to ensure the change works as expected and to prevent regressions.

1.5 Reporting a security vulnerability

If you've discovered a security vulnerability in Logical Authorization Bundle, we appreciate your help in disclosing it to us in a responsible manner.

Publicly disclosing a vulnerability can put the entire community at risk. If you've discovered a security concern, please email us at ordermind@gmail.com. We'll work with you to make sure that we understand the scope of the issue, and that we fully address your concern.

After a security vulnerability has been corrected, a security hotfix release will be deployed as soon as possible.

CHAPTER 2

Quickstart

This page provides a quick introduction to Logical Authorization Bundle and introductory examples. If you have not already installed Logical Authorization Bundle, head over to the *Installation* page.

2.1 Permission types

Permission types are used to check different kinds of conditions for access control. Listed below are the permission types available by default. You can also use boolean permissions if you want to allow or disallow an action completely. Boolean permissions are useful in combination with access bypass in some situations. Please refer to https://github.com/ordermind/logical-permissions-js#boolean-permissions for details regarding boolean permissions.

`role` Checks if a user has one or more roles.

Examples

```
// Allow access only if user has a single role

{"role": "ROLE_ADMIN"}

// Allow access only if user has at least one of the following roles

{
    "role": {
        "OR": [
            "ROLE_ADMIN",
            "ROLE_SALES"
        ]
    }
}

// Allow access only if user has both roles

{
    "role": {
        "AND": [
            "ROLE_ADMIN",
            "ROLE_SALES"
        ]
    }
}
```

flag A flag is a boolean condition that does not require any additional parameter, just a context. The following flags are available by default:

`user_can_bypass_access` checks if the user can bypass access.

`user_has_account` checks if the user has an account, i.e. is not an anonymous user.

`user_is_author` checks if the user is the author of an entity or document.

Examples

```
// Allow access if user normally can bypass access

{"flag": "user_can_bypass_access"}

// Allow access if user has an account

{"flag": "user_has_account"}

// Allow access if user is the author of an entity or document

{"flag": "user_is_author"}
```

host Checks if the current request is sent to an approved host.

Example

```
// Allow access only if the request is sent to localhost

{"host": "localhost"}
```

ip Checks if the current request comes from an approved ip address or range.

Example

```
// Allow access only if the request comes from 127.0.0.1

{"ip": "127.0.0.1"}
```

method Checks if the current request uses an approved method.

Example

```
// Allow access only if the request uses the POST method

{"method": "POST"}
```

2.2 Adding a custom permission type

Custom permission types can be added by creating a service with the tag `logauth.tag.permission_type` and which implements `Ordermind\LogicalPermissions\PermissionTypeInterface`.

If your needs are simple you may prefer to create a flag instead of a whole permission type. You can do that by creating a service with the tag `logauth.tag.permission_type.flag` and which implements `Ordermind\LogicalAuthorizationBundle\PermissionTypes\Flag\FlagInterface`.

2.3 Access Bypass

This library supports the ability for a superuser to completely circumvent access checks. To enable a user to bypass access, implement `Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface` in your user class and use the `setBypassAccess()` method. This user will now always be granted access, except if you use `NO_BYPASS` at the first level of a permission declaration.

Examples

```
// A user with bypass access enabled will be granted access no matter if they have this role or not.

{"role": "ROLE_ADMIN"}

// If you want to negate the ability to bypass access for an action, add NO_BYPASS to the first level

{
    "role": "ROLE_ADMIN",
    "NO_BYPASS": true
}
```

2.4 Declaring Permissions

Permissions may be declared both inline together with for example the declaration of a route, or in the logauth configuration file located at `/config/packages/logauth.yaml`. These permissions will override any inline permission declarations. For help with the use of logic gates and nesting, please refer to the documentation at https://github.com/ordermind/logical-permissions-js#logic-gates.

2.4.1 Routes

Inline route permission declarations are supported for routes defined with annotations, YAML and XML. You can also declare them in a configuration file, and there it is also possible to use regex patterns to declare permissions for multiple routes at once.

Annotations

YAML

XML

Config File

Note: In order to declare route permissions in annotations within a controller, the type `logauth_annotation` must be used in the routing file for this controller.

Example

If you want to enable the default application controllers for declaring permissions in the annotations, the easiest way is to find the file `/config/routes/annotations.yaml` and replace the existing type with `logauth_annotation` so that it looks like this:

```
controllers:
    resource: ../../src/Controller/
    type: logauth_annotation
```

Once your controller is configured to work with `logauth_annotation`, you may declare permissions with your route in json format by using the `@Permissions` annotation defined in `Ordermind\LogicalAuthorizationBundle\Annotation\Routing\Permissions`:

```
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Route;
use Sensio\Bundle\FrameworkExtraBundle\Configuration\Method;
use Symfony\Bundle\FrameworkBundle\Controller\Controller;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpFoundation\Response;
use Ordermind\LogicalAuthorizationBundle\Annotation\Routing\Permissions;

class DefaultController extends Controller {
    /**
      * @Route("/route-role", name="route_role")
      * @Method({"GET"})
      * @Permissions({
      *   "role": "ROLE_ADMIN"
      * })
      */
    public function routeRoleAction(Request $request) {
        return new Response('');
    }
}
```

If you define your route in yaml you can declare permissions like this:

```
route_role:
    path: /route-role
    defaults: { _controller: App\Controller\DefaultController::routeRoleAction }
    permissions:
        role: ROLE_ADMIN
```

If you define your route in xml you can declare permissions like this:

```
<route id="route_role" path="/route-role">
    <default key="_controller">App\Controller\DefaultController::routeRoleAction</default>
    <permissions>
        <role>ROLE_ADMIN</role>
    </permissions>
</route>
```

Here are a couple of examples of permission declarations that you can put in the configuration file.

Single route example

```
# LogicalAuthorization Configuration
logauth:
    permissions:
        routes:
            route_role:
                role: ROLE_ADMIN
```

Route pattern example

```
# LogicalAuthorization Configuration
logauth:
    permissions:
        route_patterns:
            ^/route-role:
                role: ROLE_ADMIN
```

2.4.2 Doctrine ORM

Inline entity permission declarations are supported for entities mapped with annotations, YAML and XML. You can also declare these permissions in the configuration file. You can declare permissions both on the entity level and on the field level. The permissions are separately declared for each action. For entities the default actions are "create", "read", "update" and "delete", while for fields they are "get" and "set".

Annotations

YAML

XML

Config File

If you map your entity with annotations, you can declare permissions in json format by using the `@Permissions` annotation defined in `Ordermind\LogicalAuthorizationDoctrineORMBundle\Annotation\Doctrine\Permission`
Here is an example of permission declarations on both entity and field levels.

```
use Doctrine\ORM\Mapping as ORM;
use Ordermind\LogicalAuthorizationDoctrineORMBundle\Annotation\Doctrine\Permissions;
use Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface;
use Ordermind\LogicalAuthorizationBundle\Interfaces\ModelInterface;

/**
* TestEntityRoleAuthor
*
* @ORM\Table(name="testentities_roleauthor")
* @ORM\Entity(repositoryClass="App\Repository\TestEntityRoleAuthorRepository")
* @Permissions({
*   "create": {
*     "role": "ROLE_ADMIN"
*   },
*   "read": {
*     "OR": {
*       "role": "ROLE_ADMIN",
*       "flag": "user_is_author"
*     }
*   },
*   "update": {
*     "OR": {
*       "role": "ROLE_ADMIN",
*       "flag": "user_is_author"
*     }
*   },
*   "delete": {
*     "OR": {
*       "role": "ROLE_ADMIN",
*       "flag": "user_is_author"
*     }
*   }
* })
*/
class TestEntityRoleAuthor implements ModelInterface
{
    /**
    * @var int
    *
    * @ORM\Column(name="id", type="integer")
    * @ORM\Id
```

```
    * @ORM\GeneratedValue(strategy="AUTO")
    */
    private $id;

    /**
    * @var string
    *
    * @ORM\Column(name="field1", type="string", length=255)
    * @Permissions({
    *   "get": {
    *     "role": "ROLE_ADMIN",
    *     "flag": "user_is_author"
    *   },
    *   "set": {
    *     "role": "ROLE_ADMIN",
    *     "flag": "user_is_author"
    *   }
    * })
    */
    private $field1 = '';

    /**
    * @var Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface
    *
    * @ORM\ManyToOne(targetEntity="App\Entity\TestUser")
    * @ORM\JoinColumn(name="author_id", referencedColumnName="id")
    */
    private $author;

    /**
    * Set field1
    *
    * @param string $field1
    *
    * @return TestEntityRoleAuthor
    */
    public function setField1($field1)
    {
        $this->field1 = $field1;

        return $this;
    }

    /**
    * Get field1
    *
    * @return string
    */
    public function getField1()
    {
        return $this->field1;
    }

    /**
    * Set author
    *
    * @param Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface $author
    *
```

```
    * @return entity implementing ModelInterface
    */
    public function setAuthor(UserInterface $author)
    {
        $this->author = $author;

        return $this;
    }

    /**
    * Get authorId
    *
    * @return Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface
    */
    public function getAuthor(): ?UserInterface
    {
        return $this->author;
    }
}
```

If you map your entity with yaml you can declare permissions like this in the mapping file:

```
App\Entity\TestEntityRoleAuthor:
    type: entity
    repositoryClass: App\Repository\TestEntityRoleAuthorRepository
    table: testentities_roleauthor

    permissions:
        create:
            role: ROLE_ADMIN
        read:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author
        update:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author
        delete:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author

    id:
        id:
            type: integer
            generator:
                strategy: AUTO

    fields:
        field1:
            type: string
            length: 255
            permissions:
                get:
                    role: ROLE_ADMIN
                    flag: user_is_author
                set:
                    role: ROLE_ADMIN
```

```
                    flag: user_is_author

    manyToOne:
        author:
            targetEntity: App\Entity\TestUser
            joinColumn:
                name: author_id
                referencedColumnName: id
```

If you map your entity with xml you can declare permissions like this in the mapping file:

```
<entity name="App\Entity\TestEntityRoleAuthor" repository-class="App\Repository\TestEntityRoleAuthorR
    <permissions>
        <create>
            <role>ROLE_ADMIN</role>
        </create>
        <read>
            <OR>
                <role>ROLE_ADMIN</role>
                <flag>user_is_author</flag>
            </OR>
        </read>
        <update>
            <OR>
                <role>ROLE_ADMIN</role>
                <flag>user_is_author</flag>
            </OR>
        </update>
        <delete>
            <OR>
                <role>ROLE_ADMIN</role>
                <flag>user_is_author</flag>
            </OR>
        </delete>
    </permissions>

    <id name="id" type="integer" column="id">
        <generator strategy="AUTO"/>
    </id>

    <field name="field1" column="field1" type="string" length="255">
        <permissions>
            <get>
                <role>ROLE_ADMIN</role>
                <flag>user_is_author</flag>
            </get>
            <set>
                <role>ROLE_ADMIN</role>
                <flag>user_is_author</flag>
            </set>
        </permissions>
    </field>

    <many-to-one field="author" target-entity="App\Entity\TestUser">
        <join-column name="author_id" referenced-column-name="id" />
    </many-to-one>
</entity>
```

In the config file you can declare entity and field permissions like this:

```
# LogicalAuthorization Configuration
logauth:
    permissions:
        models:
            App\Entity\TestEntityRoleAuthor:
                create:
                    role: ROLE_ADMIN
                read:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                update:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                delete:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                fields:
                    field1:
                        get:
                            role: ROLE_ADMIN
                            flag: user_is_author
                        set:
                            role: ROLE_ADMIN
                            flag: user_is_author
```

2.4.3 Doctrine MongoDB

Inline document permission declarations are supported for documents mapped with annotations, YAML and XML. You can also declare these permissions in a configuration file. You can declare permissions both on the document level and on the field level. The permissions are separately declared for each action. For documents the default actions are "create", "read", "update" and "delete", while for fields they are "get" and "set".

Annotations

YAML

XML

Config File

If you map your document with annotations, you can declare permissions in json format by using the `@Permissions` annotation defined in `Ordermind\LogicalAuthorizationDoctrineMongoBundle\Annotation\Doctrine\Permissions`. Here is an example of permission declarations on both document and field levels.

```
use Doctrine\ODM\MongoDB\Mapping\Annotations as ODM;
use Ordermind\LogicalAuthorizationDoctrineMongoBundle\Annotation\Doctrine\Permissions;
use Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface;
use Ordermind\LogicalAuthorizationBundle\Interfaces\ModelInterface;

/**
* TestDocumentRoleAuthor
*
* @ODM\Document(repositoryClass="App\Repository\TestDocumentRoleAuthorRepository", collection="testdo
* @Permissions({
```

```
*    "create": {
*      "role": "ROLE_ADMIN"
*    },
*    "read": {
*      "OR": {
*        "role": "ROLE_ADMIN",
*        "flag": "user_is_author"
*      }
*    },
*    "update": {
*      "OR": {
*        "role": "ROLE_ADMIN",
*        "flag": "user_is_author"
*      }
*    },
*    "delete": {
*      "OR": {
*        "role": "ROLE_ADMIN",
*        "flag": "user_is_author"
*      }
*    }
* })
*/
class TestDocumentRoleAuthor implements ModelInterface
{
    /**
    * @var int
    *
    * @ODM\Field(name="id", type="integer")
    * @ODM\Id
    */
    private $id;

    /**
    * @var string
    *
    * @ODM\Field(name="field1", type="string")
    * @Permissions({
    *    "get": {
    *      "role": "ROLE_ADMIN",
    *      "flag": "user_is_author"
    *    },
    *    "set": {
    *      "role": "ROLE_ADMIN",
    *      "flag": "user_is_author"
    *    }
    * })
    */
    private $field1 = '';

    /**
    * @var Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface
    *
    * @ODM\ReferenceOne(targetDocument="App\Document\TestUser")
    */
    protected $author;

    /**
```

```
    * Get id
    *
    * @return int
    */
    public function getId()
    {
        return $this->id;
    }

    /**
    * Set field1
    *
    * @param string $field1
    *
    * @return TestDocumentRoleAuthor
    */
    public function setField1($field1)
    {
        $this->field1 = $field1;

        return $this;
    }

    /**
    * Get field1
    *
    * @return string
    */
    public function getField1()
    {
        return $this->field1;
    }

    /**
    * Set author
    *
    * @param Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface $author
    *
    * @return TestDocumentRoleAuthor
    */
    public function setAuthor(UserInterface $author)
    {
        $this->author = $author;

        return $this;
    }

    /**
    * Get authorId
    *
    * @return Ordermind\LogicalAuthorizationBundle\Interfaces\UserInterface
    */
    public function getAuthor(): ?UserInterface
    {
        return $this->author;
    }
}
```

If you map your document with yaml you can declare permissions like this in the mapping file:

```
App\Document\TestDocumentRoleAuthor:
    type: document
    repositoryClass: App\Repository\TestDocumentRoleAuthorRepository
    collection: testdocuments_roleauthor

    permissions:
        create:
            role: ROLE_ADMIN
        read:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author
        update:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author
        delete:
            OR:
                role: ROLE_ADMIN
                flag: user_is_author

    fields:
        id:
            id: true
        field1:
            type: string
            permissions:
                get:
                    role: ROLE_ADMIN
                    flag: user_is_author
                set:
                    role: ROLE_ADMIN
                    flag: user_is_author

    referenceOne:
        author:
            targetDocument: App\Document\TestUser
```

If you map your document with xml you can declare permissions like this in the mapping file:

```
<document name="App\Document\TestDocumentRoleAuthor" repository-class="App\Repository\TestDocumentRo
  <permissions>
    <create>
      <role>ROLE_ADMIN</role>
    </create>
    <read>
      <OR>
        <role>ROLE_ADMIN</role>
        <flag>user_is_author</flag>
      </OR>
    </read>
    <update>
      <OR>
        <role>ROLE_ADMIN</role>
        <flag>user_is_author</flag>
      </OR>
    </update>
```

```
    <delete>
      <OR>
        <role>ROLE_ADMIN</role>
        <flag>user_is_author</flag>
      </OR>
    </delete>
  </permissions>

  <field name="id" id="true" />

  <field name="field1" type="string">
    <permissions>
      <get>
        <role>ROLE_ADMIN</role>
        <flag>user_is_author</flag>
      </get>
      <set>
        <role>ROLE_ADMIN</role>
        <flag>user_is_author</flag>
      </set>
    </permissions>
  </field>

  <reference-one field="author" target-document="App\Document\TestUser" />
</document>
```

In the config file you can declare document and field permissions like this:

```
# LogicalAuthorization Configuration
logauth:
    permissions:
        models:
            App\Document\TestDocumentRoleAuthor:
                create:
                    role: ROLE_ADMIN
                read:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                update:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                delete:
                    OR:
                        role: ROLE_ADMIN
                        flag: user_is_author
                fields:
                    field1:
                        get:
                            role: ROLE_ADMIN
                            flag: user_is_author
                        set:
                            role: ROLE_ADMIN
                            flag: user_is_author
```

2.5 Checking Permissions

Now that you have declared your permissions, it's time to put them to use. This section explains how you check these permissions.

2.5.1 Routes

In order to enable checking for route permissions, you need to set the following configuration in `/config/packages/security.yaml`:

```
security:
    access_control:
        - { path: ^/, allow_if: "logauth_route()" }
```

It is recommended to remove all other configuration for access_control. That way you make sure that all route permissions are handled by Logical Authorization Bundle. After setting this configuration, route permissions should work as expected.

Note: At this time there is no support for checking entity permissions automatically in routes, unless you use Sonata Admin or something similar where you can hook into their security logic. Otherwise, if you pass an entity parameter to a route and want to check those permissions to decide whether access to the route should be granted or not, you have to do that manually inside the controller. See the following section for information about how to achieve that.

To check route access in Twig, you can use the function `logauth_check_route_access` like this:

```
logauth_check_route_access(route_name)
```

2.5.2 Doctrine ORM

Entity permissions are not checked automatically. In order to make that job easier for you, there are decorators that you can use for this purpose. The recommended way is to declare each repository as a service like this:

```
repository.test_entity_roleauthor:
    class: Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Decorator\RepositoryDecorator
    factory: ['@logauth_doctrine_orm.service.repository_decorator_factory', getRepositoryDecorator]
    arguments:
        - App\Entity\TestEntityRoleAuthor
```

The only thing you will need to alter in the above configuration is the service name (`repository.test_entity_roleauthor`) and the entity class argument (`App\Entity\TestEntityRoleAuthor`) for the related entity. The rest can be left as is. When you have declared a repository as a service in this way, it can easily be fetched by dependency injection.

Note: If you need to use several repository decorators in the same class, it can be more convenient to inject the factory service `logauth_doctrine_orm.service.repository_decorator_factory` instead of each individual repository decorator service. Then you can use `Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Factory\RepositoryDecoratorFactory` to get the repository decorator that you want.

Once you have injected the repository decorator you can use it as a regular repository, but the difference is that the results from queries are automatically filtered by permissions, so if a user doesn't have permissions for the "read" action for an entity, it will be removed from the result. The resulting objects will also be instances of `Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Decorator\EntityDecorator`

rather than the raw entities. The entity decorators, just like the repository decorators, act just like the wrapped objects but allows for automatic permission checks when you try to interact with it.

One exception to this rule is lazy loaded entities, which are sometimes used for performance purposes. Because filtering would require instantiating the objects and thus defeat the whole purpose of using lazy loaded collections in the first place, these are not filtered or wrapped by decorators by default. If you do want to filter and wrap them, you can enable it like this:

```
# LogicalAuthorization Doctrine ORM Configuration
logauth_doctrine_orm:
    check_lazy_loaded_entities: true
```

If you already have an entity object and want to wrap it in a decorator to do permission checks, you can load the corresponding repository decorator service either by injecting the service for the decorator or the factory service as mentioned above. Once you have the repository decorator you can use `Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Decorator\RepositoryDecoratorInter` to get the entity decorator. This is useful if you want to check entity permissions in a controller, like this for example:

```
# File: /config/services.yaml
services:
    App\Controller\:
        resource: '../src/Controller'
        arguments: ['@logauth_doctrine_orm.service.repository_decorator_factory']
        tags: ['controller.service_arguments']
```

```
# File: /src/Controller/DefaultController.php

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Factory\RepositoryDecoratorFactoryInter
use App\Entity\TestEntityRoleAuthor;

class DefaultController extends Controller
{
    /**
    * @var Ordermind\LogicalAuthorizationDoctrineORMBundle\Services\Factory\RepositoryDecoratorFacto
    */
    private $repositoryDecoratorFactory;

    public function __construct(RepositoryDecoratorFactoryInterface $repositoryDecoratorFactory)
    {
        $this->repositoryDecoratorFactory = $repositoryDecoratorFactory;
    }

    /**
    * @Route("/testentity-role-author/view/{id}", name="testentity_role_author_view")
    */
    public function testEntityRoleAuthorViewAction(Request $request, TestEntityRoleAuthor $testEntity
    {
        // Check read access to entity and throw access denied exception if access is not granted.
        $repositoryDecorator = $this->repositoryDecoratorFactory->getRepositoryDecorator(TestEntityRo
        $entityDecorator = $repositoryDecorator->wrapEntity($testEntityRoleAuthor);
        if(!$entityDecorator->checkEntityAccess('read')) {
            throw $this->createAccessDeniedException('Read access to entity denied.');
        }

        // Code to execute if access is granted
    }
}
```

To check entity access in Twig, you can use the function `logauth_doctrine_orm_check_entity_access` like this:

```
logauth_doctrine_orm_check_entity_access(object, action)
```

To check field access in Twig, you can use the function `logauth_doctrine_orm_check_field_access` like this:

```
logauth_doctrine_orm_check_field_access(object, fieldName, action)
```

2.5.3 Doctrine MongoDB

Document permissions are not checked automatically. In order to make that job easier for you, there are decorators that you can use for this purpose. The recommended way is to declare each repository as a service like this:

```
repository.test_document_roleauthor:
    class: Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Decorator\RepositoryDecorator
    factory: ['@logauth_doctrine_mongo.service.repository_decorator_factory', getRepositoryDecorator]
    arguments:
        - App\Document\TestDocumentRoleAuthor
```

The only thing you will need to alter in the above configuration is the service name (`repository.test_document_roleauthor`) and the document class argument (`App\Document\TestDocumentRoleAuthor`) for the related document. The rest can be left as is. When you have declared a repository as a service in this way, it can easily be fetched by dependency injection.

Note: If you need to use several repository decorators in the same class, it can be more convenient to inject the factory service `logauth_doctrine_mongo.service.repository_decorator_factory` instead of each individual repository decorator service. Then you can use `Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Factory\RepositoryDecoratorFacto` to get the repository decorator that you want.

Once you have injected the repository decorator you can use it as a regular repository, but the difference is that the results from queries are automatically filtered by permissions, so if a user doesn't have permissions for the "read" action for an document, it will be removed from the result. The resulting objects will also be instances of `Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Decorator\DocumentDecorator` rather than the raw documents. The document decorators, just like the repository decorators, act just like the wrapped objects but allows for automatic permission checks when you try to interact with it.

One exception to this rule is lazy loaded documents, which are sometimes used for performance purposes. Because filtering would require instantiating the objects and thus defeat the whole purpose of using lazy loaded collections in the first place, these are not filtered or wrapped by decorators by default. If you do want to filter and wrap them, you can enable it like this:

```
# LogicalAuthorization Mongo Configuration
logauth_doctrine_mongo:
    check_lazy_loaded_documents: true
```

If you already have an document object and want to wrap it in a decorator to do permission checks, you can load the corresponding repository decorator service either by injecting the service for the decorator or the factory service as mentioned above. Once you have the repository decorator you can use `Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Decorator\RepositoryDecoratorInt` to get the document decorator. This is useful if you want to check document permissions in a controller, like this for example:

```
# File: /config/services.yaml
services:
    App\Controller\:
        resource: '../src/Controller'
        arguments: ['@logauth_doctrine_mongo.service.repository_decorator_factory']
        tags: ['controller.service_arguments']
```

```
# File: /src/Controller/DefaultController.php

use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\Routing\Annotation\Route;
use Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Factory\RepositoryDecoratorFactoryInte
use App\Document\TestDocumentRoleAuthor;

class DefaultController extends Controller
{
    /**
    * @var Ordermind\LogicalAuthorizationDoctrineMongoBundle\Services\Factory\RepositoryDecoratorFact
    */
    private $repositoryDecoratorFactory;

    public function __construct(RepositoryDecoratorFactoryInterface $repositoryDecoratorFactory)
    {
        $this->repositoryDecoratorFactory = $repositoryDecoratorFactory;
    }

    /**
    * @Route("/testentity-role-author/view/{id}", name="testentity_role_author_view")
    */
    public function testEntityRoleAuthorViewAction(Request $request, TestDocumentRoleAuthor $testDocu
    {
        // Check read access to document and throw access denied exception if access is not granted.
        $repositoryDecorator = $this->repositoryDecoratorFactory->getRepositoryDecorator(TestDocument
        $documentDecorator = $repositoryDecorator->wrapDocument($testDocumentRoleAuthor);
        if(!$documentDecorator->checkDocumentAccess('read')) {
            throw $this->createAccessDeniedException('Read access to document denied.');
        }

        // Code to execute if access is granted
    }
}
```

To check document access in Twig, you can use the function `logauth_doctrine_mongo_check_document_access` like this:

```
logauth_doctrine_mongo_check_document_access(object, action)
```

To check field access in Twig, you can use the function `logauth_doctrine_mongo_check_field_access` like this:

```
logauth_doctrine_mongo_check_field_access(object, fieldName, action)
```

2.6 Debugging

This bundle has extensive support for debugging to help making it crystal clear exactly what goes on when the permissions are being checked. If you run the environment in dev mode, you will see a padlock icon in the

devbar with the total amount of access checks for the current request, as well as their outcome. If you click on the icon you end up on the debug panel where you can take a closer look at each of the access checks to see which permissions were checked, the context and from where the access check was made. If the permissions are complex, they will be broken down into parts so that you can see the return value for each part. If you click the "Permission Tree" tab, you can navigate a tree of all the declared permissions for the site. This tree can also be seen by running the console command `logauth:dump-permission-tree`. A third way of getting the permission tree is to use the service `logauth.service.permission_tree_builder` and call `Ordermind\LogicalAuthorizationBundle\Services\PermissionTreeBuilderInterface::getTree()`. This can be useful if you want to expose the permission tree via an API.

2.6.1 Erroneous Permissions

If you make a mistake and declare permissions that are not syntactically valid, an exception will be thrown when these permissions are checked except if the environment is set to "prod". In that case it will be quietly logged and the access check will fail.

2.6.2 Caching

The whole permissions tree is cached for performance reasons, so if you change the permissions you'll need to make sure that the cache is cleared before the updated permissions work. In order to disable caching for debugging purposes, you can use a NULL adapter for Symfony's caching system. Here's how you do that.

1. Add the following service to `/config/services.yml`:

```
services:
    cache.adapter.null:
        class: Symfony\Component\Cache\Adapter\NullAdapter
        abstract: true
        arguments: [~, ~, ~]
        tags:
            - {name: cache.pool, clearer: cache.default_clearer}
```

2. Use this configuration in `config/packages/framework.yaml`:

```
framework:
    cache:
        app: cache.adapter.null
```