Nextcloud Developer Manual

Release 11 alpha

The Nextcloud developers

December 05, 2017

CHAPTER

ONE

TABLE OF CONTENTS

General Contributor Guidelines

Community Code of Conduct

Preamble:

In the Nextcloud community, participants from all over the world come together to create Free Software for a free internet. This is made possible by the support, hard work and enthusiasm of thousands of people, including those who create and use Nextcloud software.

This document offers some guidance to ensure Nextcloud participants can cooperate effectively in a positive and inspiring atmosphere, and to explain how together we can strengthen and support each other.

This Code of Conduct is shared by all contributors and users who engage with the Nextcloud team and its community services.

Overview

This Code of Conduct presents a summary of the shared values and "common sense" thinking in our community. The basic social ingredients that hold our project together include:

- Be considerate
- Be respectful
- Be collaborative
- Be pragmatic
- Support others in the community
- Get support from others in the community

Our community is made up of several groups of individuals and organizations which can roughly be divided into two groups:

- Contributors, or those who add value to the project through improving Nextcloud software and its services
- Users, or those who add value to the project through their support as consumers of Nextcloud software

This Code of Conduct reflects the agreed standards of behavior for members of the Nextcloud community, in any forum, mailing list, wiki, web site, IRC channel, public meeting or private correspondence within the context of the Nextcloud team and its services. The community acts according to the standards written down in this Code of Conduct and will defend these standards for the benefit of the community. Leaders of any group, such as moderators of mailing

lists, IRC channels, forums, etc., will exercise the right to suspend access to any person who persistently breaks our shared Code of Conduct.

Be considerate

Your actions and work will affect and be used by other people and you in turn will depend on the work and actions of others. Any decision you take will affect other community members, and we expect you to take those consequences into account when making decisions.

As a contributor, ensure that you give full credit for the work of others and bear in mind how your changes affect others. It is also expected that you try to follow the development schedule and guidelines.

As a user, remember that contributors work hard on their part of Nextcloud and take great pride in it. If you are frustrated your problems are more likely to be resolved if you can give accurate and well-mannered information to all concerned.

Be respectful

In order for the Nextcloud community to stay healthy its members must feel comfortable and accepted. Treating one another with respect is absolutely necessary for this. In a disagreement, in the first instance assume that people mean well.

We do not tolerate personal attacks, racism, sexism or any other form of discrimination. Disagreement is inevitable, from time to time, but respect for the views of others will go a long way to winning respect for your own view. Respecting other people, their work, their contributions and assuming well-meaning motivation will make community members feel comfortable and safe and will result in motivation and productivity.

We expect members of our community to be respectful when dealing with other contributors, users and communities. Remember that Nextcloud is an international project and that you may be unaware of important aspects of other cultures.

Be collaborative

The Free Software Movement depends on collaboration: it helps limit duplication of effort while improving the quality of the software produced. In order to avoid misunderstanding, try to be clear and concise when requesting help or giving it. Remember it is easy to misunderstand emails (especially when they are not written in your mother tongue). Ask for clarifications if unsure how something is meant; remember the first rule – assume in the first instance that people mean well.

As a contributor, you should aim to collaborate with other community members, as well as with other communities that are interested in or depend on the work you do. Your work should be transparent and be fed back into the community when available, not just when Nextcloud releases. If you wish to work on something new in existing projects, keep those projects informed of your ideas and progress.

It may not always be possible to reach consensus on the implementation of an idea, so don't feel obliged to achieve this before you begin. However, always ensure that you keep the outside world informed of your work, and publish it in a way that allows outsiders to test, discuss and contribute to your efforts.

Contributors on every project come and go. When you leave or disengage from the project, in whole or in part, you should do so with pride about what you have achieved and by acting responsibly towards others who come after you to continue the project.

As a user, your feedback is important, as is its form. Poorly thought out comments can cause pain and the demotivation of other community members, but considerate discussion of problems can bring positive results. An encouraging word works wonders.

Be pragmatic

Nextcloud is a pragmatic community. We value tangible results over having the last word in a discussion. We defend our core values like freedom and respectful collaboration, but we don't let arguments about minor issues get in the way of achieving more important results. We are open to suggestions and welcome solutions regardless of their origin. When in doubt support a solution which helps getting things done over one which has theoretical merits, but isn't being worked on. Use the tools and methods which help getting the job done. Let decisions be taken by those who do the work.

Support others in the community

Our community is made strong by mutual respect, collaboration and pragmatic, responsible behavior. Sometimes there are situations where this has to be defended and other community members need help.

If you witness others being attacked, think first about how you can offer them personal support. If you feel that the situation is beyond your ability to help individually, go privately to the victim and ask if some form of official intervention is needed. Similarly you should support anyone who appears to be in danger of burning out, either through work-related stress or personal problems.

When problems do arise, consider respectfully reminding those involved of our shared Code of Conduct as a first action. Leaders are defined by their actions, and can help set a good example by working to resolve issues in the spirit of this Code of Conduct before they escalate.

Get support from others in the community

Disagreements, both political and technical, happen all the time. Our community is no exception to the rule. The goal is not to avoid disagreements or differing views but to resolve them constructively. You should turn to the community to seek advice and to resolve disagreements and where possible consult the team most directly involved.

Think deeply before turning a disagreement into a public dispute. If necessary request mediation, trying to resolve differences in a less highly-emotional medium. If you do feel that you or your work is being attacked, take your time to breathe through before writing heated replies. Consider a 24 hour moratorium if emotional language is being used – a cooling off period is sometimes all that is needed. If you really want to go a different way, then we encourage you to publish your ideas and your work, so that it can be tried and tested.

The authors of this document would like to thank the Nextcloud community and those who have worked to create such a dynamic environment to share in and who offered their thoughts and wisdom in the authoring of this document. We would also like to thank other vibrant communities that have helped shape this document with their own examples, especially KDE.

Development Environment

Please follow the steps on this page to set up your development environment.

Set up Web server and database

First set up your Web server and database (**Section**: Manual Installation - Prerequisites).

Get the source

There are two ways to obtain Nextcloud sources:

- Using the stable version
- Using the development version from GitHub which will be explained below.

To check out the source from GitHub you will need to install git (see Setting up git from the GitHub help)

Gather information about server setup

To get started the basic git repositories need to cloned into the Web server's directory. Depending on the distribution this will either be

- **/var/www**
- **/var/www/html**
- **/srv/http**

Then identify the user and group the Web server is running as and the Apache user and group for the **chown** command will either be

- **http**
- **www-data**
- **apache**
- **wwwrun**

Check out the code

The following commands are using **/var/www** as the Web server's directory and **www-data** as user name and group.

After the development tool installation make the directory writable:

```
sudo chmod o+rw /var/www
```

Then install Nextcloud from git:

```
git clone git@github.com:nextcloud/server.git /var/www/<folder>
cd /var/www/<folder>
git submodule update --init
```

where <folder> is the folder where you want to install Nextcloud.

Adjust rights:

```
sudo chown -R www-data:www-data /var/www/core/data/
sudo chmod o-rw /var/www
```

Finally restart the Web server (this might vary depending on your distribution):

```
sudo systemctl restart httpd.service
```

or:

```
sudo /etc/init.d/apache2 restart
```

After the clone Open http://localhost/core (or the corresponding URL) in your web browser to set up your instance.

Enabling debug mode

Note: Do not enable this for production! This can create security problems and is only meant for debugging and development!

To disable JavaScript and CSS caching debugging has to be enabled by setting `debug` to `true` in `config/config.php`:

```
<?php
$CONFIG = array (
    'debug' => true,
    ... configuration goes here ...
);
```

Keep the code up-to-date

If you have more than one repository cloned, it can be time consuming to do the same the action to all repositories one by one. To solve this, you can use the following command template:

```
find . -maxdepth <DEPTH> -type d -name .git -exec sh -c 'cd "{}"/../ && pwd && <GIT COMMAND>' \;
```

then, e.g. to pull all changes in all repositories, you only need this:

```
find . -maxdepth 3 -type d -name .git -exec sh -c 'cd "{}"/../ && pwd && git pull --rebase' \;
```

or to prune all merged branches, you would execute this:

```
find . -maxdepth 3 -type d -name .git -exec sh -c 'cd "{}"/../ && pwd && git remote prune origin' \;
```

It is even easier if you create alias from these commands in case you want to avoid retyping those each time you need them.

Security Guidelines

This guideline highlights some of the most common security problems and how to prevent them. Please review your app if it contains any of the following security holes.

Note: **Program defensively**: for instance always check for CSRF or escape strings, even if you do not need it. This prevents future problems where you might miss a change that leads to a security hole.

Note: All App Framework security features depend on the call of the controller through `OCA\AppFramework\App::main`. If the controller method is executed directly, no security checks are being performed!

SQL Injection

SQL Injection occurs when SQL query strings are concatenated with variables.

To prevent this, always use prepared queries:

```
<?php
$sql = 'SELECT * FROM `users` WHERE `id` = ?';
$query = \OCP\DB::prepare($sql);
$params = array(1);
$result = $query->execute($params);
```

If the App Framework is used, write SQL queries like this in the a class that extends the Mapper:

```
<?php
// inside a child mapper class
$sql = 'SELECT * FROM `users` WHERE `id` = ?';
$params = array(1);
$result = $this->execute($sql, $params);
```

Cross site scripting

Cross site scripting happens when user input is passed directly to templates. A potential attacker might be able to inject HTML/JavaScript into the page to steal the users session, log keyboard entries, even perform DDOS attacks on other websites or other malicious actions.

Despite the fact that Nextcloud uses Content-Security-Policy to prevent the execution of inline JavaScript code developers are still required to prevent XSS. CSP is just another layer of defense that is not implemented in all web browsers.

To prevent XSS in your app you have to sanitize the templates and all JavaScripts which performs a DOM manipulation.

Templates

Let's assume you use the following example in your application:

```
<?php
echo $_GET['username'];
```

An attacker might now easily send the user a link to:

```
app.php?username=<script src="attacker.tld"></script>
```

to overtake the user account. The same problem occurs when outputting content from the database or any other location that is writable by users.

Another attack vector that is often overlooked is XSS in **href** attributes. HTML allows to execute javascript in href attributes like this:

```
<a href="javascript:alert('xss')">
```

To prevent XSS in your app, **never use echo, print() or <%=** - use **p()** instead which will sanitize the input. Also **validate URLs to start with the expected protocol** (starts with http for instance)!

Note: Should you ever require to print something unescaped, double check if it is really needed. If there is no other way (e.g. when including of subtemplates) use *print_unescaped* with care.

JavaScript

Avoid manipulating the HTML directly via JavaScript, this often leads to XSS since people often forget to sanitize variables:

```
var html = '<li>' + username + '</li>"';
```

If you **really** want to use JavaScript for something like this use *escapeHTML* to sanitize the variables:

```
var html = '<li>' + escapeHTML(username) + '</li>';
```

An even better way to make your app safer is to use the jQuery built-in function **$.text()** instead of **$.html()**.

DON'T

```
messageTd.html(username);
```

DO

```
messageTd.text(username);
```

It may also be wise to choose a proper JavaScript framework like AngularJS which automatically handles the JavaScript escaping for you.

Clickjacking

Clickjacking tricks the user to click into an invisible iframe to perform an arbitrary action (e.g. delete an user account)

To prevent such attacks Nextcloud sends the *X-Frame-Options* header to all template responses. Don't remove this header if you don't really need it!

This is already built into Nextcloud if `OC_Template`.

Code executions / File inclusions

Code Execution means that an attacker is able to include an arbitrary PHP file. This PHP file runs with all the privileges granted to the normal application and can do an enormous amount of damage.

Code executions and file inclusions can be easily prevented by **never** allowing user-input to run through the following functions:

- **include()**
- **require()**
- **require_once()**
- **eval()**
- **fopen()**

Note: Also **never** allow the user to upload files into a folder which is reachable from the URL!

DON'T

```
<?php
require("/includes/" . $_GET['file']);
```

Note: If you have to pass user input to a potentially dangerous function, double check to be sure that there is no other way. If it is not possible otherwise sanitize every user parameter and ask people to audit your sanitize function.

Directory Traversal

Very often developers forget about sanitizing the file path (removing all \ and /), this allows an attacker to traverse through directories on the server which opens several potential attack vendors including privilege escalations, code executions or file disclosures.

DON'T

```
<?php
$username = OC_User::getUser();
fopen("/data/" . $username . "/" . $_GET['file'] . ".txt");
```

DO

```
<?php
$username = OC_User::getUser();
$file = str_replace(array('/', '\\'), '',  $_GET['file']);
fopen("/data/" . $username . "/" . $file . ".txt");
```

Note: PHP also interprets the backslash (\) in paths, don't forget to replace it too!

Shell Injection

Shell Injection occurs if PHP code executes shell commands (e.g. running a latex compiler). Before doing this, check if there is a PHP library that already provides the needed functionality. If you really need to execute a command be aware that you have to escape every user parameter passed to one of these functions:

- **exec()**
- **shell_exec()**
- **passthru()**
- **proc_open()**
- **system()**
- **popen()**

Note: Please require/request additional programmers to audit your escape function.

Without escaping the user input this will allow an attacker to execute arbitrary shell commands on your server.

PHP offers the following functions to escape user input:

- **escapeshellarg()**: Escape a string to be used as a shell argument
- **escapeshellcmd()**: Escape shell metacharacters

DON'T

```
<?php
system('ls '.$_GET['dir']);
```

DO

```
<?php
system('ls '.escapeshellarg($_GET['dir']));
```

Auth bypass / Privilege escalations

Auth bypass/privilege escalations happen when a user is able to perform unauthorized actions.

Nextcloud offers three simple checks:

- **OCP\JSON::checkLoggedIn()**: Checks if the logged in user is logged in
- **OCP\JSON::checkAdminUser()**: Checks if the logged in user has admin privileges
- **OCP\JSON::checkSubAdminUser()**: Checks if the logged in user has group admin privileges

Using the App Framework, these checks are already automatically performed for each request and have to be explicitly turned off by using annotations above your controller method, see Controllers.

Additionally always check if the user has the right to perform that action. (e.g. a user should not be able to delete other users' bookmarks).

Sensitive data exposure

Always store user data or configuration files in safe locations, e.g. **nextcloud/data/** and not in the webroot where they can be accessed by anyone using a web browser.

Cross site request forgery

Using CSRF one can trick a user into executing a request that he did not want to make. Thus every POST and GET request needs to be protected against it. The only places where no CSRF checks are needed are in the main template, which is rendering the application, or in externally callable interfaces.

Note: Submitting a form is also a POST/GET request!

To prevent CSRF in an app, be sure to call the following method at the top of all your files:

```
<?php
OCP\JSON::callCheck();
```

If you are using the App Framework, every controller method is automatically checked for CSRF unless you explicitly exclude it by setting the @NoCSRFRequired annotation before the controller method, see Controllers

Unvalidated redirects

This is more of an annoyance than a critical security vulnerability since it may be used for social engineering or phishing.

Always validate the URL before redirecting if the requested URL is on the same domain or an allowed resource.

DON'T

```
<?php
header('Location:'. $_GET['redirectURL']);
```

DO

```
<?php
header('Location: https://example.com'. $_GET['redirectURL']);
```

Getting help

If you need help to ensure that a function is secure please ask on our mailing list or on our IRC channel **#nextcloud-dev** on **irc.freenode.net**.

Coding Style & General Guidelines

General

- Ideally, discuss your plans on the forums to see if others want to work with you on it
- We use Github, please get an account there and clone the repositories you want to work on
- Fixes go directly to master, nevertheless they need to be tested thoroughly.
- New features are always developed in a branch and only merged to master once they are fully done.
- Software should work. We only put features into master when they are complete. It's better to not have a feature instead of having one that works poorly.
- It is best to start working based on an issue - create one if there is none. You describe what you want to do, ask feedback on the direction you take it and take it from there.
- When you are finished, use the merge request function on Github to create a pull request. The other developers will look at it and give you feedback. You can signify that your PR is ready for review by adding the label "5 - ready for review" to it. You can also post your merge request to the mailing list to let people know. See the code review page for more information
- It is key to keep changes separate and small. The bigger and more hairy a PR grows, the harder it is to get it in. So split things up where you can in smaller changes - if you need a small improvement like a API addition for a big feature addition, get it in first rather than adding it to the big piece of work!
- Decisions are made by consensus. We strive for making the best technical decisions and as nobody can know everything, we collaborate. That means a first negative comment might not be the final word, neither is positive feedback an immediate GO. Nextcloud is built out of modular pieces (apps) and maintainers have a strong influence. In case of disagreement we consult other seasoned contributors.

Labels

We assign labels to issues and pull requests to make it easy to find them and to signal what needs to be done. Some of these are assigned by the developers, others by QA, bug triagers, project lead or maintainers and so on. It is not desired that users/reporters of bugs assign labels themselves, unless they are developers/contributors to Nextcloud.

The most important labels and their meaning:

- #backport-request - the pull requests also needs to be applied to older Nextcloud versions
- #bug - this issue is a bug
- #enhancement - this issue is a feature request/idea for improvement of Nextcloud

- #design - this needs help from the design team or is a design-related issue/pull request
- #technical debt - this issue or PR is about technical debt
- #starter issue - these are issues which are relatively easy to solve and ideal for people who want to learn how to code in Nextcloud
- #needs info - this issue needs further information from the reporter, see triaging
- #high #medium #low signify how important the bug is.
- Tags showing the state of the issue or PR, numbered 0-4: * #0 - Needs triage - ready to start development on this * #1 - To develop - ready to start development on this * #2 - Developing - development in progress * #3 - To Review - ready for review * #4 - To Release - reviewed PR that awaits unfreeze of a branch to get merged
- Feature tags: #feature: something. These tags indicate the features across apps and components which are impacted by the issue or which the PR is related to

If you want a label not in the list above, please first discuss on the mailing list.

Coding

- Maximum line-length of 80 characters
- Use tabs to indent
- A tab is 4 spaces wide
- Opening braces of blocks are on the same line as the definition
- Quotes: ‘ for everything, ” for HTML attributes (<p class=”my_class”>)
- End of Lines : Unix style (LF / ‘n’) only
- No global variables or functions
- Unit tests
- HTML should be HTML5 compliant
- Check these database performance tips
- When you `git pull`, always `git pull --rebase` to avoid generating extra commits like: *merged master into master*

License Headers

Nextcloud is licensed under the GNU AGPLv3. From June, 16 2016 on we switched to “GNU AGPLv3 or any later version” for better long-term maintainability. If you create a new file please use this header:

```
/**
 *
 * @copyright Copyright (c) <year>, <your name> (<your email address>)
 *
 * @license GNU AGPL version 3 or any later version
 *
 * This program is free software: you can redistribute it and/or modify
 * it under the terms of the GNU Affero General Public License as
 * published by the Free Software Foundation, either version 3 of the
 * License, or (at your option) any later version.
 *
 * This program is distributed in the hope that it will be useful,
```

```
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU Affero General Public License for more details.
 *
 * You should have received a copy of the GNU Affero General Public License
 * along with this program.  If not, see <http://www.gnu.org/licenses/>.
 *
 */
```

If you edit an existing file please add a copyright notice with your name, if you consider your changes substantial enough to claim copyright. As a rule if thumb, this is the case if you contributed more than seven lines of code.

User interface

- Software should get out of the way. Do things automatically instead of offering configuration options.
- Software should be easy to use. Show only the most important elements. Secondary elements only on hover or via Advanced function.
- User data is sacred. Provide undo instead of asking for confirmation - which might be dismissed
- The state of the application should be clear. If something loads, provide feedback.
- Do not adapt broken concepts (for example design of desktop apps) just for the sake of consistency. We aim to provide a better interface, so let's find out how to do that!
- Regularly reset your installation to see how the first-run experience is like. And improve it.
- Ideally do usability testing to know how people use the software.
- For further UX principles, read Alex Faaborg from Mozilla.

PHP

The Nextcloud coding style guide is based on PEAR Coding Standards.

Always use:

```
<?php
```

at the start of your php code. The final closing:

```
?>
```

should not be used at the end of the file due to the possible issue of sending white spaces.

Comments

All API methods need to be marked with PHPDoc markup. An example would be:

```
<?php

/**
 * Description what method does
 * @param Controller $controller the controller that will be transformed
 * @param API $api an instance of the API class
 * @throws APIException if the api is broken
 * @since 4.5
```

```
 * @return string a name of a user
 */
public function myMethod(Controller $controller, API $api) {
  // ...
}
```

Objects, Functions, Arrays & Variables

Use Pascal case for Objects, Camel case for functions and variables. If you set a default function/method parameter, do not use spaces. Do not prepend private class members with underscores.

```
class MyClass {

}

function myFunction($default=null) {

}

$myVariable = 'blue';

$someArray = array(
    'foo'  => 'bar',
    'spam' => 'ham',
);

?>
```

Operators

Use **===** and **!==** instead of **==** and **!=**.

Here's why:

```
<?php

var_dump(0 == "a"); // 0 == 0 -> true
var_dump("1" == "01"); // 1 == 1 -> true
var_dump("10" == "1e1"); // 10 == 10 -> true
var_dump(100 == "1e2"); // 100 == 100 -> true

?>
```

Control Structures

- Always use { } for one line ifs
- Split long ifs into multiple lines
- Always use break in switch statements and prevent a default block with warnings if it shouldn't be accessed

```
<?php

// single line if
if ($myVar === 'hi') {
```

```
    $myVar = 'ho';
} else {
    $myVar = 'bye';
}

// long ifs
if (   $something === 'something'
    || $condition2
    && $condition3
) {
  // your code
}

// for loop
for ($i = 0; $i < 4; $i++) {
    // your code
}

switch ($condition) {
    case 1:
        // action1
        break;

    case 2:
        // action2;
        break;

    default:
        // defaultaction;
        break;
}

?>
```

Unit tests

Unit tests must always extend the `\Test\TestCase` class, which takes care of cleaning up the installation after the test.

If a test is run with multiple different values, a data provider must be used. The name of the data provider method must not start with `test` and must end with `Data`.

```
<?php
namespace Test;
class Dummy extends \Test\TestCase {
    public function dummyData() {
        return array(
            array(1, true),
            array(2, false),
        );
    }

    /**
     * @dataProvider dummyData
     */
    public function testDummy($input, $expected) {
        $this->assertEquals($expected, \Dummy::method($input));
```

```
    }
}
```

JavaScript

In general take a look at JSLint without the whitespace rules.

- Use a `js/main.js` or `js/app.js` where your program is started
- Complete every statement with a **;**
- Use **var** to limit variable to local scope
- To keep your code local, wrap everything in a self executing function. To access global objects or export things to the global namespace, pass all global objects to the self executing function.
- Use JavaScript strict mode
- Use a global namespace object where you bind publicly used functions and objects to

DO:

```
// set up namespace for sharing across multiple files
var MyApp = MyApp || {};

(function(window, $, exports, undefined) {
    'use strict';

    // if this function or object should be global, attach it to the namespace
    exports.myGlobalFunction = function(params) {
        return params;
    };

})(window, jQuery, MyApp);
```

DONT (Seriously):

```
// This does not only make everything global but you're programming
// JavaScript like C functions with namespaces
MyApp = {
    myFunction:function(params) {
        return params;
    },
    ...
};
```

Objects & Inheritance

Try to use OOP in your JavaScript to make your code reusable and flexible.

This is how you'd do inheritance in JavaScript:

```
// create parent object and bind methods to it
var ParentObject = function(name) {
    this.name = name;
};

ParentObject.prototype.sayHello = function() {
    console.log(this.name);
```

```
}


// create childobject, call parents constructor and inherit methods
var ChildObject = function(name, age) {
    ParentObject.call(this, name);
    this.age = age;
};

ChildObject.prototype = Object.create(ParentObject.prototype);

// overwrite parent method
ChildObject.prototype.sayHello = function() {
    // call parent method if you want to
    ParentObject.prototype.sayHello.call(this);
    console.log('childobject');
};

var child = new ChildObject('toni', 23);

// prints:
// toni
// childobject
child.sayHello();
```

Objects, Functions & Variables

Use Pascal case for Objects, Camel case for functions and variables.

```
var MyObject = function() {
    this.attr = "hi";
};

var myFunction = function() {
    return true;
};

var myVariable = 'blue';

var objectLiteral = {
    value1: 'somevalue'
};
```

Operators

Use **===** and **!==** instead of **==** and **!=**.

Here's why:

```
'' == '0'           // false
0 == ''             // true
0 == '0'            // true

false == 'false'    // false
false == '0'        // true
```

```
false == undefined  // false
false == null       // false
null == undefined   // true

' \t\r\n ' == 0     // true
```

Control Structures

- Always use { } for one line ifs
- Split long ifs into multiple lines
- Always use break in switch statements and prevent a default block with warnings if it shouldn't be accessed

DO:

```
// single line if
if (myVar === 'hi') {
    myVar = 'ho';
} else {
    myVar = 'bye';
}

// long ifs
if (   something === 'something'
    || condition2
    && condition3
) {
  // your code
}

// for loop
for (var i = 0; i < 4; i++) {
    // your code
}

// switch
switch (value) {

    case 'hi':
        // yourcode
        break;

    default:
        console.warn('Entered undefined default block in switch');
        break;
}
```

CSS

Take a look at the Writing Tactical CSS & HTML video on YouTube.

Don't bind your CSS too much to your HTML structure and try to avoid IDs. Also try to make your CSS reusable by grouping common attributes into classes.

DO:

```
.list {
    list-style-type: none;
}

.list > .list_item {
    display: inline-block;
}

.important_list_item {
    color: red;
}
```

DON'T:

```
#content .myHeader ul {
    list-style-type: none;
}

#content .myHeader ul li.list_item {
    color: red;
    display: inline-block;
}
```

TBD

Performance Considerations

This document introduces some common considerations and tips on improving performance of Nextcloud. Speed of Nextcloud is important - nobody likes to wait and often, what is *just slow* for a small amount of data will become *unusable* with a large amount of data. Please keep these tips in mind when developing for Nextcloud and consider reviewing your app to make it faster.

Database performance

The database plays an important role in Nextcloud performance. The general rule is: database queries are very bad and should be avoided if possible. The reasons for that are:

- Roundtrips: Bigger Nextcloud installations have the database not installed on the application server but on a remote dedicated database server. The problem is that database queries then go over the network. These roundtrips can add up significantly if you have a lot of queries.
- Speed. A lot of people think that databases are fast. This is not always true if you compare it with handling data internally in PHP or in the filesystem or even using key/value based storages. So every developer should always double check if the database is really the best place for the data.
- Scalability. If you have a big Nextcloud cluster setup you usually have several Nextcloud/Web servers in parallel and a central database and a central storage. This means that everything that happens on the Nextcloud/PHP side can parallelize and can be scaled. Stuff that is happening in the database and in the storage is critical because it only exists once and can't be scaled so easily.

We can reduce the load on the database by:

1. Making sure that every query uses an index.
2. Reducing the overall number of queries.
3. If you are familiar with cache invalidation you can try caching query results in PHP.

There a several ways to monitor which queries are actually executed on the database.

With MySQL it is very easy with just a bit of configuration:

1. Slow query log.

If you put this into your my.cnf file, every query that takes longer than one second is logged to a logfile:

```
log_slow_queries = 1
log_slow_queries = /var/log/mysql/mysql-slow.log
long_query_time=1
```

If a query takes more than a second we have a serious problem of course. You can watch it with *tail -f /var/log/mysql/mysql-slow.log* while using Nextcloud.

2. log all queries.

If you reduce the long_query_time to zero then every statement is logged. This is super helpful to see what is going on. Just do a *tail -f* on the logfile and click around in the interface or access the WebDAV interface:

```
log_slow_queries = 1
log_slow_queries = /var/log/mysql/mysql-slow.log
long_query_time=0
```

3. log queries without an index.

If you increase the long_query_time to 100 and add log-queries-not-using-indexes, all the queries that are not using an index are logged. Every query should always use an index. So ideally there should be no output:

```
log-queries-not-using-indexes
log_slow_queries = 1
log_slow_queries = /var/log/mysql/mysql-slow.log
long_query_time=100
```

Measuring performance

If you do bigger changes in the architecture or the database structure you should always double check the positive or negative performance impact. There are a few nice small scripts that can be used for this.

The recommendation is to automatically do 10000 PROPFINDs or file uploads, measure the time and compare the time before and after the change.

Getting help

If you need help with performance or other issues please ask on our forums or on our IRC channel **#nextcloud-dev** on **irc.freenode.net**.

Debugging

Debug mode

When debug mode is enabled in Nextcloud, a variety of debugging features are enabled - see debugging documentation. Set `debug` to `true` in `/config/config.php` to enable it:

```
<?php
$CONFIG = array (
    'debug' => true,
    ... configuration goes here ...
);
```

Identifying errors

Nextcloud uses custom error PHP handling that prevents errors being printed to Web server log files or command line output. Instead, errors are generally stored in Nextcloud's own log file, located at: `/data/nextcloud.log`

Debugging variables

You should use exceptions if you need to debug variable values manually, and not alternatives like trigger_error() (which may not be logged).

e.g.:

```
<?php throw new \Exception( "\$user = $user" ); // should be logged in Nextcloud ?>
```

not:

```
<?php trigger_error( "\$user = $user" ); // may not be logged anywhere ?>
```

To disable custom error handling in Nextcloud (and have PHP and your Web server handle errors instead), see Debug mode.

Using a PHP debugger (XDebug)

Using a debugger connected to PHP allows you to step through code line by line, view variables at each line and even change values while the code is running. The de-facto standard debugger for PHP is XDebug, available as an installable package in many distributions. It just provides the PHP side however, so you will need a frontend to actually control XDebug. When installed, it needs to be enabled in `php.ini`, along with some parameters to enable connections to the debugging interface:

```
zend_extension=/usr/lib/php/modules/xdebug.so
xdebug.remote_enable=on
xdebug.remote_host=127.0.0.1
xdebug.remote_port=9000
xdebug.remote_handler=dbgp
```

XDebug will now (when activated) try to connect to localhost on port 9000, and will communicate over the standard protocol DBGP. This protocol is supported by many debugging interfaces, such as the following popular ones:

- vdebug - Multi-language DBGP debugger client for Vim
- SublimeTextXdebug - XDebug client for Sublime Text
- PHPStorm - in-built DBGP debugger

For further reading, see the XDebug documentation: http://xdebug.org/docs/remote

Once you are familiar with how your debugging client works, you can start debugging with XDebug. To test Nextcloud through the web interface or other HTTP requests, set the `XDEBUG_SESSION_START` cookie or POST parameter. Alternatively, there are browser extensions to make this easy:

- The Easiest XDebug (Firefox): https://addons.mozilla.org/en-US/firefox/addon/the-easiest-xdebug/

- XDebug Helper (Chrome): https://chrome.google.com/extensions/detail/eadndfjplgieldjbigjakmdgkmoaaaoc

For debugging scripts on the command line, like `occ` or unit tests, set the `XDEBUG_CONFIG` environment variable.

Debugging Javascript

By default all Javascript files in Nextcloud are minified (compressed) into a single file without whitespace. To prevent this, see Debug mode.

Debugging HTML and templates

By default Nextcloud caches HTML generated by templates. This may prevent changes to app templates, for example, from being applied on page refresh. To disable caching, see Debug mode.

Using alternative app directories

It may be useful to have multiple app directories for testing purposes, so you can conveniently switch between different versions of applications. See the configuration file documentation for details.

Backporting

General

We backport important fixes and improvements from the current master release to get them to our users faster.

Process

We mostly consider bug fixes for back porting. Occasionally, important changes to the API can be backported to make it easier for developers to keep their apps working between major releases. If you think a pull request (PR) is relevant for the stable release, go through these steps:

1. Make sure the PR is merged to master
2. Ask Frank (**@karlitschek**), if the code should be backported and add the label backport-request to the PR
3. If Frank approves, create a new branch based on the respective stable branch (stable10 for the 10.0.x series), cherry-pick the needed commits to that branch and create a PR on GitHub.
4. Specify the corresponding milestone for that series (10.0.x-next-maintenance for the 10.0.x series) to this PR and reference the original PR in there. This enables the QA team to find the backported items for testing and having the original PR with detailed description linked.

Note: Before each patch release there is a freeze to be able to test everything as a whole without pulling in new changes. While this freeze is active a backport isn't allowed and has to wait for the next patch release.

The QA team will try to reproduce all the issues with the X.Y.Z-next-maintenance milestone on the relevant release and verify it is fixed by the patch release (and doesn't cause new problems). Once the patch release is out, the post-fix -next-maintenance is removed and a new -next-maintenance milestone is created for that series.

Changelog

Deprecations

This is a deprecation roadmap which lists all current deprecation targets and will be updated from release to release. This lists the year when a specific method or class will be removed.

Note: Deprecations on interfaces also affect the implementing classes!

2018

- **OCP\App::setActiveNavigationEntry** has been deprecated in favour of **\OCP\INavigationManager**
- **OCP\BackgroundJob::registerJob** has been deprecated in favour of **OCP\BackgroundJob\IJobList**
- **OCP\Contacts** functions has been deprecated in favour of **\OCP\Contacts\IManager**
- **OCP\DB** functions have been deprecated in favour of the ones in **\OCP\IDBConnection**
- **OCP\Files::tmpFile** has been deprecated in favour of **\OCP\ITempManager::getTemporaryFile**
- **OCP\Files::tmpFolder** has been deprecated in favour of **\OCP\ITempManager::getTemporaryFolder**
- **\OCP\IServerContainer::getDb** has been deprecated in favour of **\OCP\IServerContainer::getDatabaseConnection**
- **\OCP\IServerContainer::getHTTPHelper** has been deprecated in favour of **\OCP\Http\Client\IClientService**
- Legacy applications not using the AppFramework are now likely to use the deprecated **OCP\JSON** and **OCP\Response** code:
  - **\OCP\JSON** has been completely deprecated in favour of the AppFramework. Developers shall use the AppFramework instead of using the legacy **OCP\JSON** code. This allows testable controllers and is highly encouraged.
  - **\OCP\Response** has been completely deprecated in favour of the AppFramework. Developers shall use the AppFramework instead of using the legacy **OCP\JSON** code. This allows testable controllers and is highly encouraged.
- Diverse **OCP\Users** function got deprecated in favour of **OCP\IUserManager**:
  - **OCP\Users::getUsers** has been deprecated in favour of **OCP\IUserManager::search**
  - **OCP\Users::getDisplayName** has been deprecated in favour of **OCP\IUserManager::getDisplayName**
  - **OCP\Users::getDisplayNames** has been deprecated in favour of **OCP\IUserManager::searchDisplayName**
  - **OCP\Users::userExists** has been deprecated in favour of **OCP\IUserManager::userExists**
- Various static **OCP\Util** functions have been deprecated:
  - **OCP\Util::linkToRoute** has been deprecated in favour of **\OCP\IURLGenerator::linkToRoute**
  - **OCP\Util::linkTo** has been deprecated in favour of **\OCP\IURLGenerator::linkTo**
  - **OCP\Util::imagePath** has been deprecated in favour of **\OCP\IURLGenerator::imagePath**
  - **OCP\Util::isValidPath** has been deprecated in favour of **\OCP\IURLGenerator::imagePath**
- OCP\AppFramework\IAppContainer: methods **getCoreApi** and **log**

- OCP\AppFramework\IApi: full class

2017

- **OCP\IDb**: This interface and the implementing classes will be removed in favor of **OCP\IDbConnection**. Various layers in between have also been removed to be consistent with the PDO classes. This leads to the following changes:
- Replace all calls on the db using **getInsertId** with **lastInsertId**
- Replace all calls on the db using **prepareQuery** with **prepare**
- The **__construct** method of **OCP\AppFramework\Db\Mapper** no longer requires an instance of **OCP\IDb** but an instance of **OCP\IDbConnection**
- The **execute** method on **OCP\AppFramework\Db\Mapper** no longer returns an instance of **OC_DB_StatementWrapper** but an instance of **PDOStatement**

2016

- The following methods have been moved into the **OCP\Template::<method>** class instead of being namespaced directly:
- **OCP\image_path**
- **OCP\mimetype_icon**
- **OCP\preview_icon**
- **OCP\publicPreview_icon**
- **OCP\human_file_size**
- **OCP\relative_modified_date**
- **OCP\html_select_options**
- **OCP\simple_file_size** has been deprecated in favour of **OCP\Template::human_file_size**
- The **OCP\PERMISSION_<permission>** and **OCP\FILENAME_INVALID_CHARS** have been moved to **OCP\Constants::<old name>**
- The **OC_GROUP_BACKEND_<method>** and **OC_USER_BACKEND_<method>** have been moved to **OC_Group_Backend::<method>** and **OC_User_Backend::<method>** respectively
- OCP\AppFramework\Controller: methods **params**, **getParams**, **method**, **getUploadedFile**, **env**, **cookie**, **render**

2015

- \OC\Preferences and \OC_Preferences

Tutorial

This tutorial will outline how to create a very simple notes app. The finished app is available on GitHub.

Setup

First the development environment needs to be set up. This can be done by either downloading the zip from the website or cloning it directly from GitHub:

```
git clone git@github.com:nextcloud/server.git --branch $BRANCH
git submodule update --init
```

Note: `$BRANCH` is the desired Nextcloud branch (e.g. `stable9` for Nextcloud 9, `stable10` for Nextcloud 10, ..., `master` for the upcoming release)

First you want to enable debug mode to get proper error messages. To do that set `debug` to `true` in the **nextcloud/config/config.php** file:

```
<?php
$CONFIG = array (
    'debug' => true,
    ... configuration goes here ...
);
```

Note: PHP errors are logged to **nextcloud/data/nextcloud.log**

Now open another terminal window and start the development server:

```
cd nextcloud
php -S localhost:8080
```

Afterwards a skeleton app can be created in the app store.

Download the extracted the downloaded file and move it into your `apps/` directory. Afterwards the application can be enabled on the apps page.

The first basic app is now available at `http://localhost:8080/index.php/apps/yourappid/`

Routes & Controllers

A typical web application consists of server side and client side code. The glue between those two parts are the URLs. In case of the notes app the following URLs will be used:

- **GET /**: Returns the interface in HTML
- **GET /notes**: Returns a list of all notes in JSON
- **GET /notes/1**: Returns a note with the id 1 in JSON
- **DELETE /notes/1**: Deletes a note with the id 1
- **POST /notes**: Creates a new note by passing in JSON
- **PUT /notes/1**: Updates a note with the id 1 by passing in JSON

On the client side we can call these URLs with the following jQuery code:

```
// example for calling the PUT /notes/1 URL
var baseUrl = OC.generateUrl('/apps/ownnotes');
var note = {
    title: 'New note',
```

```
    content: 'This is the note text'
};
var id = 1;
$.ajax({
    url: baseUrl + '/notes/' + id,
    type: 'PUT',
    contentType: 'application/json',
    data: JSON.stringify(note)
}).done(function (response) {
    // handle success
}).fail(function (response, code) {
    // handle failure
});
```

On the server side we need to register a callback that is executed once the request comes in. The callback itself will be a method on a controller and the controller will be connected to the URL with a route. The controller and route for the page are already set up in **ownnotes/appinfo/routes.php**:

```
<?php
return ['routes' => [
    ['name' => 'page#index', 'url' => '/', 'verb' => 'GET']
]];
```

This route calls the controller **OCA\OwnNotes\PageController->index()** method which is defined in **ownnotes/lib/Controller/PageController.php**. The controller returns a template, in this case **ownnotes/templates/main.php**:

Note: @NoAdminRequired and @NoCSRFRequired in the comments above the method turn off security checks, see Controllers

---

```
<?php
 namespace OCA\OwnNotes\Controller;

 use OCP\IRequest;
 use OCP\AppFramework\Http\TemplateResponse;
 use OCP\AppFramework\Controller;

 class PageController extends Controller {

     public function __construct($AppName, IRequest $request){
         parent::__construct($AppName, $request);
     }

     /**
      * @NoAdminRequired
      * @NoCSRFRequired
      */
     public function index() {
         return new TemplateResponse('ownnotes', 'main');
     }

 }
```

Since the route which returns the initial HTML has been taken care of, the controller which handles the AJAX requests for the notes needs to be set up. Create the following file: **ownnotes/lib/Controller/NoteController.php** with the following content:

```php
<?php
 namespace OCA\OwnNotes\Controller;

 use OCP\IRequest;
 use OCP\AppFramework\Controller;

 class NoteController extends Controller {

     public function __construct($AppName, IRequest $request){
         parent::__construct($AppName, $request);
     }

     /**
      * @NoAdminRequired
      */
     public function index() {
         // empty for now
     }

     /**
      * @NoAdminRequired
      *
      * @param int $id
      */
     public function show($id) {
         // empty for now
     }

     /**
      * @NoAdminRequired
      *
      * @param string $title
      * @param string $content
      */
     public function create($title, $content) {
         // empty for now
     }

     /**
      * @NoAdminRequired
      *
      * @param int $id
      * @param string $title
      * @param string $content
      */
     public function update($id, $title, $content) {
         // empty for now
     }

     /**
      * @NoAdminRequired
      *
      * @param int $id
      */
     public function destroy($id) {
         // empty for now
     }
```

```
}
```

Note: The parameters are extracted from the request body and the url using the controller method's variable names. Since PHP does not support type hints for primitive types such as ints and booleans, we need to add them as annotations in the comments. In order to type cast a parameter to an int, add **@param int $parameterName**

Now the controller methods need to be connected to the corresponding URLs in the **ownnotes/appinfo/routes.php** file:

```
<?php
return [
    'routes' => [
        ['name' => 'page#index', 'url' => '/', 'verb' => 'GET'],
        ['name' => 'note#index', 'url' => '/notes', 'verb' => 'GET'],
        ['name' => 'note#show', 'url' => '/notes/{id}', 'verb' => 'GET'],
        ['name' => 'note#create', 'url' => '/notes', 'verb' => 'POST'],
        ['name' => 'note#update', 'url' => '/notes/{id}', 'verb' => 'PUT'],
        ['name' => 'note#destroy', 'url' => '/notes/{id}', 'verb' => 'DELETE']
    ]
];
```

Since those 5 routes are so common, they can be abbreviated by adding a resource instead:

```
<?php
return [
    'resources' => [
        'note' => ['url' => '/notes']
    ],
    'routes' => [
        ['name' => 'page#index', 'url' => '/', 'verb' => 'GET']
    ]
];
```

Database

Now that the routes are set up and connected the notes should be saved in the database. To do that first create a database schema by creating **ownnotes/appinfo/database.xml**:

```
<database>
    <name>*dbname*</name>
    <create>true</create>
    <overwrite>false</overwrite>
    <charset>utf8</charset>
    <table>
        <name>*dbprefix*ownnotes_notes</name>
        <declaration>
            <field>
                <name>id</name>
                <type>integer</type>
                <notnull>true</notnull>
                <autoincrement>true</autoincrement>
                <unsigned>true</unsigned>
                <primary>true</primary>
                <length>8</length>
            </field>
```

```
            <field>
                <name>title</name>
                <type>text</type>
                <length>200</length>
                <default></default>
                <notnull>true</notnull>
            </field>
            <field>
                <name>user_id</name>
                <type>text</type>
                <length>200</length>
                <default></default>
                <notnull>true</notnull>
            </field>
            <field>
                <name>content</name>
                <type>clob</type>
                <default></default>
                <notnull>true</notnull>
            </field>
        </declaration>
    </table>
</database>
```

To create the tables in the database, the version tag in **ownnotes/appinfo/info.xml** needs to be increased:

```
<?xml version="1.0"?>
<info>
    <id>ownnotes</id>
    <name>Own Notes</name>
    <description>My first Nextcloud app</description>
    <licence>AGPL</licence>
    <author>Your Name</author>
    <version>0.0.2</version>
    <namespace>OwnNotes</namespace>
    <category>tool</category>
    <dependencies>
        <owncloud min-version="8" />
    </dependencies>
</info>
```

Reload the page to trigger the database migration.

Now that the tables are created we want to map the database result to a PHP object to be able to control data. First create an entity in **ownnotes/lib/Db/Note.php**:

```
<?php
namespace OCA\OwnNotes\Db;

use JsonSerializable;

use OCP\AppFramework\Db\Entity;

class Note extends Entity implements JsonSerializable {

    protected $title;
    protected $content;
    protected $userId;
```

```
    public function jsonSerialize() {
        return [
            'id' => $this->id,
            'title' => $this->title,
            'content' => $this->content
        ];
    }
}
```

Note: A field **id** is automatically set in the Entity base class

We also define a **jsonSerializable** method and implement the interface to be able to transform the entity to JSON easily.

Entities are returned from so called Mappers. Let's create one in **ownnotes/lib/Db/NoteMapper.php** and add a **find** and **findAll** method:

```
<?php
namespace OCA\OwnNotes\Db;

use OCP\IDbConnection;
use OCP\AppFramework\Db\Mapper;

class NoteMapper extends Mapper {

    public function __construct(IDbConnection $db) {
        parent::__construct($db, 'ownnotes_notes', '\OCA\OwnNotes\Db\Note');
    }

    public function find($id, $userId) {
        $sql = 'SELECT * FROM *PREFIX*ownnotes_notes WHERE id = ? AND user_id = ?';
        return $this->findEntity($sql, [$id, $userId]);
    }

    public function findAll($userId) {
        $sql = 'SELECT * FROM *PREFIX*ownnotes_notes WHERE user_id = ?';
        return $this->findEntities($sql, [$userId]);
    }

}
```

Note: The first parent constructor parameter is the database layer, the second one is the database table and the third is the entity on which the result should be mapped onto. Insert, delete and update methods are already implemented.

Connect Database & Controllers

The mapper which provides the database access is finished and can be passed into the controller.

You can pass in the mapper by adding it as a type hinted parameter. Nextcloud will figure out how to assemble them by itself. Additionally we want to know the userId of the currently logged in user. Simply add a **$UserId** parameter to the constructor (case sensitive!). To do that open **ownnotes/lib/Controller/NoteController.php** and change it to the following:

```php
<?php
namespace OCA\OwnNotes\Controller;

use Exception;

use OCP\IRequest;
use OCP\AppFramework\Http;
use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\Controller;

use OCA\OwnNotes\Db\Note;
use OCA\OwnNotes\Db\NoteMapper;

class NoteController extends Controller {

    private $mapper;
    private $userId;

    public function __construct($AppName, IRequest $request, NoteMapper $mapper, $UserId){
        parent::__construct($AppName, $request);
        $this->mapper = $mapper;
        $this->userId = $UserId;
    }

    /**
     * @NoAdminRequired
     */
    public function index() {
        return new DataResponse($this->mapper->findAll($this->userId));
    }

    /**
     * @NoAdminRequired
     *
     * @param int $id
     */
    public function show($id) {
        try {
            return new DataResponse($this->mapper->find($id, $this->userId));
        } catch(Exception $e) {
            return new DataResponse([], Http::STATUS_NOT_FOUND);
        }
    }

    /**
     * @NoAdminRequired
     *
     * @param string $title
     * @param string $content
     */
    public function create($title, $content) {
        $note = new Note();
        $note->setTitle($title);
        $note->setContent($content);
        $note->setUserId($this->userId);
        return new DataResponse($this->mapper->insert($note));
    }
```

```
    /**
     * @NoAdminRequired
     *
     * @param int $id
     * @param string $title
     * @param string $content
     */
    public function update($id, $title, $content) {
        try {
            $note = $this->mapper->find($id, $this->userId);
        } catch(Exception $e) {
            return new DataResponse([], Http::STATUS_NOT_FOUND);
        }
        $note->setTitle($title);
        $note->setContent($content);
        return new DataResponse($this->mapper->update($note));
    }

    /**
     * @NoAdminRequired
     *
     * @param int $id
     */
    public function destroy($id) {
        try {
            $note = $this->mapper->find($id, $this->userId);
        } catch(Exception $e) {
            return new DataResponse([], Http::STATUS_NOT_FOUND);
        }
        $this->mapper->delete($note);
        return new DataResponse($note);
    }

}
```

Note: The actual exceptions are **OCP\AppFramework\Db\DoesNotExistException** and **OCP\AppFramework\Db\MultipleObjectsReturnedException** but in this example we will treat them as the same. DataResponse is a more generic response than JSONResponse and also works with JSON.

This is all that is needed on the server side. Now let's progress to the client side.

Making things reusable and decoupling controllers from the database

Let's say our app is now on the app store and and we get a request that we should save the files in the filesystem which requires access to the filesystem.

The filesystem API is quite different from the database API and throws different exceptions, which means we need to rewrite everything in the **NoteController** class to use it. This is bad because a controller's only responsibility should be to deal with incoming Http requests and return Http responses. If we need to change the controller because the data storage was changed the code is probably too tightly coupled and we need to add another layer in between. This layer is called **Service**.

Let's take the logic that was inside the controller and put it into a separate class inside **owncloud/lib/Service/NoteService.php**:

```php
<?php
namespace OCA\OwnNotes\Service;

use Exception;

use OCP\AppFramework\Db\DoesNotExistException;
use OCP\AppFramework\Db\MultipleObjectsReturnedException;

use OCA\OwnNotes\Db\Note;
use OCA\OwnNotes\Db\NoteMapper;


class NoteService {

    private $mapper;

    public function __construct(NoteMapper $mapper){
        $this->mapper = $mapper;
    }

    public function findAll($userId) {
        return $this->mapper->findAll($userId);
    }

    private function handleException ($e) {
        if ($e instanceof DoesNotExistException ||
            $e instanceof MultipleObjectsReturnedException) {
            throw new NotFoundException($e->getMessage());
        } else {
            throw $e;
        }
    }

    public function find($id, $userId) {
        try {
            return $this->mapper->find($id, $userId);

        // in order to be able to plug in different storage backends like files
        // for instance it is a good idea to turn storage related exceptions
        // into service related exceptions so controllers and service users
        // have to deal with only one type of exception
        } catch(Exception $e) {
            $this->handleException($e);
        }
    }

    public function create($title, $content, $userId) {
        $note = new Note();
        $note->setTitle($title);
        $note->setContent($content);
        $note->setUserId($userId);
        return $this->mapper->insert($note);
    }

    public function update($id, $title, $content, $userId) {
        try {
            $note = $this->mapper->find($id, $userId);
            $note->setTitle($title);
```

```
            $note->setContent($content);
            return $this->mapper->update($note);
        } catch(Exception $e) {
            $this->handleException($e);
        }
    }

    public function delete($id, $userId) {
        try {
            $note = $this->mapper->find($id, $userId);
            $this->mapper->delete($note);
            return $note;
        } catch(Exception $e) {
            $this->handleException($e);
        }
    }

}
```

Following up create the exceptions in **ownnotes/lib/Service/ServiceException.php**:

```
<?php
namespace OCA\OwnNotes\Service;

use Exception;

class ServiceException extends Exception {}
```

and **ownnotes/lib/Service/NotFoundException.php**:

```
<?php
namespace OCA\OwnNotes\Service;

class NotFoundException extends ServiceException {}
```

Remember how we had all those ugly try catches that where checking for **DoesNotExistException** and simply returned a 404 response? Let's also put this into a reusable class. In our case we chose a trait so we can inherit methods without having to add it to our inheritance hierarchy. This will be important later on when you've got controllers that inherit from the **ApiController** class instead.

The trait is created in **ownnotes/lib/Controller/Errors.php**:

```
<?php

namespace OCA\OwnNotes\Controller;

use Closure;

use OCP\AppFramework\Http;
use OCP\AppFramework\Http\DataResponse;

use OCA\OwnNotes\Service\NotFoundException;


trait Errors {

    protected function handleNotFound (Closure $callback) {
        try {
            return new DataResponse($callback());
```

```
        } catch(NotFoundException $e) {
            $message = ['message' => $e->getMessage()];
            return new DataResponse($message, Http::STATUS_NOT_FOUND);
        }
    }

}
```

Now we can wire up the trait and the service inside the **NoteController**:

```
<?php
namespace OCA\OwnNotes\Controller;

use OCP\IRequest;
use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\Controller;

use OCA\OwnNotes\Service\NoteService;

class NoteController extends Controller {

    private $service;
    private $userId;

    use Errors;

    public function __construct($AppName, IRequest $request,
                                NoteService $service, $UserId){
        parent::__construct($AppName, $request);
        $this->service = $service;
        $this->userId = $UserId;
    }

    /**
     * @NoAdminRequired
     */
    public function index() {
        return new DataResponse($this->service->findAll($this->userId));
    }

    /**
     * @NoAdminRequired
     *
     * @param int $id
     */
    public function show($id) {
        return $this->handleNotFound(function () use ($id) {
            return $this->service->find($id, $this->userId);
        });
    }

    /**
     * @NoAdminRequired
     *
     * @param string $title
     * @param string $content
     */
    public function create($title, $content) {
```

```
        return $this->service->create($title, $content, $this->userId);
    }

    /**
     * @NoAdminRequired
     *
     * @param int $id
     * @param string $title
     * @param string $content
     */
    public function update($id, $title, $content) {
        return $this->handleNotFound(function () use ($id, $title, $content) {
            return $this->service->update($id, $title, $content, $this->userId);
        });
    }

    /**
     * @NoAdminRequired
     *
     * @param int $id
     */
    public function destroy($id) {
        return $this->handleNotFound(function () use ($id) {
            return $this->service->delete($id, $this->userId);
        });
    }

}
```

Great! Now the only reason that the controller needs to be changed is when request/response related things change.

Writing a test for the controller (recommended)

Tests are essential for having happy users and a carefree life. No one wants their users to rant about your app breaking their Nextcloud or being buggy. To do that you need to test your app. Since this amounts to a ton of repetitive tasks, we need to automate the tests.

Unit Tests

A unit test is a test that tests a class in isolation. It is very fast and catches most of the bugs, so we want many unit tests.

Because Nextcloud uses Dependency Injection to assemble your app, it is very easy to write unit tests by passing mocks into the constructor. A simple test for the update method can be added by adding this to **ownnotes/tests/Unit/Controller/NoteControllerTest.php**:

```
<?php
namespace OCA\OwnNotes\Tests\Unit\Controller;

use PHPUnit_Framework_TestCase;

use OCP\AppFramework\Http;
use OCP\AppFramework\Http\DataResponse;

use OCA\OwnNotes\Service\NotFoundException;
```

```
class NoteControllerTest extends PHPUnit_Framework_TestCase {

    protected $controller;
    protected $service;
    protected $userId = 'john';
    protected $request;

    public function setUp() {
        $this->request = $this->getMockBuilder('OCP\IRequest')->getMock();
        $this->service = $this->getMockBuilder('OCA\OwnNotes\Service\NoteService')
            ->disableOriginalConstructor()
            ->getMock();
        $this->controller = new NoteController(
            'ownnotes', $this->request, $this->service, $this->userId
        );
    }

    public function testUpdate() {
        $note = 'just check if this value is returned correctly';
        $this->service->expects($this->once())
            ->method('update')
            ->with($this->equalTo(3),
                    $this->equalTo('title'),
                    $this->equalTo('content'),
                   $this->equalTo($this->userId))
            ->will($this->returnValue($note));

        $result = $this->controller->update(3, 'title', 'content');

        $this->assertEquals($note, $result->getData());
    }


    public function testUpdateNotFound() {
        // test the correct status code if no note is found
        $this->service->expects($this->once())
            ->method('update')
            ->will($this->throwException(new NotFoundException()));

        $result = $this->controller->update(3, 'title', 'content');

        $this->assertEquals(Http::STATUS_NOT_FOUND, $result->getStatus());
    }

}
```

We can and should also create a test for the **NoteService** class:

```
<?php
namespace OCA\OwnNotes\Tests\Unit\Service;

use PHPUnit_Framework_TestCase;

use OCP\AppFramework\Db\DoesNotExistException;

use OCA\OwnNotes\Db\Note;

class NoteServiceTest extends PHPUnit_Framework_TestCase {
```

```
    private $service;
    private $mapper;
    private $userId = 'john';

    public function setUp() {
        $this->mapper = $this->getMockBuilder('OCA\OwnNotes\Db\NoteMapper')
            ->disableOriginalConstructor()
            ->getMock();
        $this->service = new NoteService($this->mapper);
    }

    public function testUpdate() {
        // the existing note
        $note = Note::fromRow([
            'id' => 3,
            'title' => 'yo',
            'content' => 'nope'
        ]);
        $this->mapper->expects($this->once())
            ->method('find')
            ->with($this->equalTo(3))
            ->will($this->returnValue($note));

        // the note when updated
        $updatedNote = Note::fromRow(['id' => 3]);
        $updatedNote->setTitle('title');
        $updatedNote->setContent('content');
        $this->mapper->expects($this->once())
            ->method('update')
            ->with($this->equalTo($updatedNote))
            ->will($this->returnValue($updatedNote));

        $result = $this->service->update(3, 'title', 'content', $this->userId);

        $this->assertEquals($updatedNote, $result);
    }


    /**
     * @expectedException OCA\OwnNotes\Service\NotFoundException
     */
    public function testUpdateNotFound() {
        // test the correct status code if no note is found
        $this->mapper->expects($this->once())
            ->method('find')
            ->with($this->equalTo(3))
            ->will($this->throwException(new DoesNotExistException('')));

        $this->service->update(3, 'title', 'content', $this->userId);
    }

}
```

If PHPUnit is installed we can run the tests inside **ownnotes/** with the following command:

```
phpunit
```

Note: You need to adjust the **ownnotes/tests/Unit/Controller/PageControllerTest** file to get the tests passing: remove the **testEcho** method since that method is no longer present in your **PageController** and do not test the user id parameters since they are not passed anymore

Integration Tests

Integration tests are slow and need a fully working instance but make sure that our classes work well together. Instead of mocking out all classes and parameters we can decide whether to use full instances or replace certain classes. Because they are slow we don't want as many integration tests as unit tests.

In our case we want to create an integration test for the udpate method without mocking out the **NoteMapper** class so we actually write to the existing database.

To do that create a new file called **ownnotes/tests/Integration/NoteIntegrationTest.php** with the following content:

```
<?php
namespace OCA\OwnNotes\Tests\Integration\Controller;

use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\App;
use Test\TestCase;

use OCA\OwnNotes\Db\Note;

class NoteIntregrationTest extends TestCase {

    private $controller;
    private $mapper;
    private $userId = 'john';

    public function setUp() {
        parent::setUp();
        $app = new App('ownnotes');
        $container = $app->getContainer();

        // only replace the user id
        $container->registerService('UserId', function($c) {
            return $this->userId;
        });

        $this->controller = $container->query(
            'OCA\OwnNotes\Controller\NoteController'
        );

        $this->mapper = $container->query(
            'OCA\OwnNotes\Db\NoteMapper'
        );
    }

    public function testUpdate() {
        // create a new note that should be updated
        $note = new Note();
        $note->setTitle('old_title');
```

```
        $note->setContent('old_content');
        $note->setUserId($this->userId);

        $id = $this->mapper->insert($note)->getId();

        // fromRow does not set the fields as updated
        $updatedNote = Note::fromRow([
            'id' => $id,
            'user_id' => $this->userId
        ]);
        $updatedNote->setContent('content');
        $updatedNote->setTitle('title');

        $result = $this->controller->update($id, 'title', 'content');

        $this->assertEquals($updatedNote, $result->getData());

        // clean up
        $this->mapper->delete($result->getData());
    }

}
```

To run the integration tests change into the **ownnotes** directory and run:

```
phpunit -c phpunit.integration.xml
```

Adding a RESTful API (optional)

A RESTful API allows other apps such as Android or iPhone apps to access and change your notes. Since syncing is a big core component of Nextcloud it is a good idea to add (and document!) your own RESTful API.

Because we put our logic into the **NoteService** class it is very easy to reuse it. The only pieces that need to be changed are the annotations which disable the CSRF check (not needed for a REST call usually) and add support for CORS so your API can be accessed from other webapps.

With that in mind create a new controller in **ownnotes/lib/Controller/NoteApiController.php**:

```
<?php
namespace OCA\OwnNotes\Controller;

use OCP\IRequest;
use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\ApiController;

use OCA\OwnNotes\Service\NoteService;

class NoteApiController extends ApiController {

    private $service;
    private $userId;

    use Errors;

    public function __construct($AppName, IRequest $request,
                                NoteService $service, $UserId){
        parent::__construct($AppName, $request);
        $this->service = $service;
```

```
        $this->userId = $UserId;
    }

    /**
     * @CORS
     * @NoCSRFRequired
     * @NoAdminRequired
     */
    public function index() {
        return new DataResponse($this->service->findAll($this->userId));
    }

    /**
     * @CORS
     * @NoCSRFRequired
     * @NoAdminRequired
     *
     * @param int $id
     */
    public function show($id) {
        return $this->handleNotFound(function () use ($id) {
            return $this->service->find($id, $this->userId);
        });
    }

    /**
     * @CORS
     * @NoCSRFRequired
     * @NoAdminRequired
     *
     * @param string $title
     * @param string $content
     */
    public function create($title, $content) {
        return $this->service->create($title, $content, $this->userId);
    }

    /**
     * @CORS
     * @NoCSRFRequired
     * @NoAdminRequired
     *
     * @param int $id
     * @param string $title
     * @param string $content
     */
    public function update($id, $title, $content) {
        return $this->handleNotFound(function () use ($id, $title, $content) {
            return $this->service->update($id, $title, $content, $this->userId);
        });
    }

    /**
     * @CORS
     * @NoCSRFRequired
     * @NoAdminRequired
     *
     * @param int $id
```

```
     */
    public function destroy($id) {
        return $this->handleNotFound(function () use ($id) {
            return $this->service->delete($id, $this->userId);
        });
    }

}
```

All that is left is to connect the controller to a route and enable the built in preflighted CORS method which is defined in the **ApiController** base class:

```
<?php
return [
    'resources' => [
        'note' => ['url' => '/notes'],
        'note_api' => ['url' => '/api/0.1/notes']
    ],
    'routes' => [
        ['name' => 'page#index', 'url' => '/', 'verb' => 'GET'],
        ['name' => 'note_api#preflighted_cors', 'url' => '/api/0.1/{path}',
         'verb' => 'OPTIONS', 'requirements' => ['path' => '.+']]
    ]
];
```

Note: It is a good idea to version your API in your URL

You can test the API by running a GET request with **curl**:

```
curl -u user:password http://localhost:8080/index.php/apps/ownnotes/api/0.1/notes
```

Since the **NoteApiController** is basically identical to the **NoteController**, the unit test for it simply inherits its tests from the **NoteControllerTest**. Create the file **ownnotes/tests/Unit/Controller/NoteApiControllerTest.php**:

```
<?php
namespace OCA\OwnNotes\Tests\Unit\Controller;

require_once __DIR__ . '/NoteControllerTest.php';

class NoteApiControllerTest extends NoteControllerTest {

    public function setUp() {
        parent::setUp();
        $this->controller = new NoteApiController(
            'ownnotes', $this->request, $this->service, $this->userId
        );
    }

}
```

Adding JavaScript and CSS

To create a modern webapp you need to write JavaScript. You can use any JavaScript framework but for this tutorial we want to keep it as simple as possible and therefore only include the templating library handlebarsjs. Download the file into **ownnotes/js/handlebars.js** and include it at the very top of **ownnotes/templates/main.php** before the other scripts and styles:

```
<?php
script('ownnotes', 'handlebars');
```

Note: jQuery is included by default on every page.

Creating a navigation

The template file **ownnotes/templates/part.navigation.php** contains the navigation. Nextcloud defines many handy CSS styles which we are going to reuse to style the navigation. Adjust the file to contain only the following code:

Note: **$l->t()** is used to make your strings translatable and **p()** is used to print escaped HTML

```
<!-- translation strings -->
<div style="display:none" id="new-note-string"><?php p($l->t('New note')); ?></div>

<script id="navigation-tpl" type="text/x-handlebars-template">
    <li id="new-note"><a href="#"><?php p($l->t('Add note')); ?></a></li>
    {{#each notes}}
        <li class="note with-menu {{#if active}}active{{/if}}"  data-id="{{ id }}">
            <a href="#">{{ title }}</a>
            <div class="app-navigation-entry-utils">
                <ul>
                    <li class="app-navigation-entry-utils-menu-button svg"><button></button></li>
                </ul>
            </div>

            <div class="app-navigation-entry-menu">
                <ul>
                    <li><button class="delete icon-delete svg" title="delete"></button></li>
                </ul>
            </div>
        </li>
    {{/each}}
</script>

<ul></ul>
```

Creating the content

The template file **ownnotes/templates/part.content.php** contains the content. It will just be a textarea and a button, so replace the content with the following:

```
<script id="content-tpl" type="text/x-handlebars-template">
    {{#if note}}
        <div class="input"><textarea>{{ note.content }}</textarea></div>
        <div class="save"><button><?php p($l->t('Save')); ?></button></div>
    {{else}}
        <div class="input"><textarea disabled></textarea></div>
        <div class="save"><button disabled><?php p($l->t('Save')); ?></button></div>
    {{/if}}
</script>
<div id="editor"></div>
```

Wiring it up

When the page is loaded we want all the existing notes to load. Furthermore we want to display the current note when you click on it in the navigation, a note should be deleted when we click the deleted button and clicking on **New note** should create a new note. To do that open **ownnotes/js/script.js** and replace the example code with the following:

```
(function (OC, window, $, undefined) {
'use strict';

$(document).ready(function () {

var translations = {
    newNote: $('#new-note-string').text()
};

// this notes object holds all our notes
var Notes = function (baseUrl) {
    this._baseUrl = baseUrl;
    this._notes = [];
    this._activeNote = undefined;
};

Notes.prototype = {
    load: function (id) {
        var self = this;
        this._notes.forEach(function (note) {
            if (note.id === id) {
                note.active = true;
                self._activeNote = note;
            } else {
                note.active = false;
            }
        });
    },
    getActive: function () {
        return this._activeNote;
    },
    removeActive: function () {
        var index;
        var deferred = $.Deferred();
        var id = this._activeNote.id;
        this._notes.forEach(function (note, counter) {
            if (note.id === id) {
                index = counter;
            }
        });

        if (index !== undefined) {
            // delete cached active note if necessary
            if (this._activeNote === this._notes[index]) {
                delete this._activeNote;
            }

            this._notes.splice(index, 1);

            $.ajax({
                url: this._baseUrl + '/' + id,
                method: 'DELETE'
```

```
            }).done(function () {
                deferred.resolve();
            }).fail(function () {
                deferred.reject();
            });
        } else {
            deferred.reject();
        }
        return deferred.promise();
    },
    create: function (note) {
        var deferred = $.Deferred();
        var self = this;
        $.ajax({
            url: this._baseUrl,
            method: 'POST',
            contentType: 'application/json',
            data: JSON.stringify(note)
        }).done(function (note) {
            self._notes.push(note);
            self._activeNote = note;
            self.load(note.id);
            deferred.resolve();
        }).fail(function () {
            deferred.reject();
        });
        return deferred.promise();
    },
    getAll: function () {
        return this._notes;
    },
    loadAll: function () {
        var deferred = $.Deferred();
        var self = this;
        $.get(this._baseUrl).done(function (notes) {
            self._activeNote = undefined;
            self._notes = notes;
            deferred.resolve();
        }).fail(function () {
            deferred.reject();
        });
        return deferred.promise();
    },
    updateActive: function (title, content) {
        var note = this.getActive();
        note.title = title;
        note.content = content;

        return $.ajax({
            url: this._baseUrl + '/' + note.id,
            method: 'PUT',
            contentType: 'application/json',
            data: JSON.stringify(note)
        });
    }
};

// this will be the view that is used to update the html
```

```
var View = function (notes) {
    this._notes = notes;
};

View.prototype = {
    renderContent: function () {
        var source = $('#content-tpl').html();
        var template = Handlebars.compile(source);
        var html = template({note: this._notes.getActive()});

        $('#editor').html(html);

        // handle saves
        var textarea = $('#app-content textarea');
        var self = this;
        $('#app-content button').click(function () {
            var content = textarea.val();
            var title = content.split('\n')[0]; // first line is the title

            self._notes.updateActive(title, content).done(function () {
                self.render();
            }).fail(function () {
                alert('Could not update note, not found');
            });
        });
    },
    renderNavigation: function () {
        var source = $('#navigation-tpl').html();
        var template = Handlebars.compile(source);
        var html = template({notes: this._notes.getAll()});

        $('#app-navigation ul').html(html);

        // create a new note
        var self = this;
        $('#new-note').click(function () {
            var note = {
                title: translations.newNote,
                content: ''
            };

            self._notes.create(note).done(function() {
                self.render();
                $('#editor textarea').focus();
            }).fail(function () {
                alert('Could not create note');
            });
        });

        // show app menu
        $('#app-navigation .app-navigation-entry-utils-menu-button').click(function () {
            var entry = $(this).closest('.note');
            entry.find('.app-navigation-entry-menu').toggleClass('open');
        });

        // delete a note
        $('#app-navigation .note .delete').click(function () {
            var entry = $(this).closest('.note');
```

```
            entry.find('.app-navigation-entry-menu').removeClass('open');

            self._notes.removeActive().done(function () {
                self.render();
            }).fail(function () {
                alert('Could not delete note, not found');
            });
        });

        // load a note
        $('#app-navigation .note > a').click(function () {
            var id = parseInt($(this).parent().data('id'), 10);
            self._notes.load(id);
            self.render();
            $('#editor textarea').focus();
        });
    },
    render: function () {
        this.renderNavigation();
        this.renderContent();
    }
};

var notes = new Notes(OC.generateUrl('/apps/ownnotes/notes'));
var view = new View(notes);
notes.loadAll().done(function () {
    view.render();
}).fail(function () {
    alert('Could not load notes');
});


});

})(OC, window, jQuery);
```

Apply finishing touches

Now the only thing left is to style the textarea in a nicer fashion. To do that open **ownnotes/css/style.css** and replace the content with the following CSS code:

```
#app-content-wrapper {
    height: 100%;
}

#editor {
    height: 100%;
    width: 100%;
}

#editor .input {
    height: calc(100% - 51px);
    width: 100%;
}

#editor .save {
    height: 50px;
```

```
    width: 100%;
    text-align: center;
    border-top: 1px solid #ccc;
    background-color: #fafafa;
}

#editor textarea {
    height: 100%;
    width: 100%;
    border: 0;
    margin: 0;
    border-radius: 0;
    overflow-y: auto;
}

#editor button {
    height: 44px;
}
```

Congratulations! You've written your first Nextcloud app. You can now either try to further improve the tutorial notes app or start writing your own app.

Create an app

After you've set up the development environment change into the Nextcloud apps directory:

```
cd /var/www/nextcloud/apps
```

Then create a skeleton app in the app store.

Enable the app

The app can now be enabled on the Nextcloud apps page

App architecture

The following directories have now been created:

- **appinfo/**: Contains app metadata and configuration
- **css/**: Contains the CSS
- **js/**: Contains the JavaScript files
- **lib/**: Contains the php class files of your app
- **templates/**: Contains the templates
- **tests/**: Contains the tests

Navigation and Pre-App configuration

The `appinfo/app.php` is the first file that is loaded and executed in Nextcloud. Depending on the purpose of the app it usually just contains the navigation setup, and maybe Background Jobs (Cron) and Hooks registrations. This is

how an example `appinfo/app.php` could look like:

```
<?php

\OC::$server->getNavigationManager()->add(function () {
    $urlGenerator = \OC::$server->getURLGenerator();
    return [
        // the string under which your app will be referenced in nextcloud
        'id' => 'myapp',

        // sorting weight for the navigation. The higher the number, the higher
        // will it be listed in the navigation
        'order' => 10,

        // the route that will be shown on startup
        'href' => $urlGenerator->linkToRoute('myapp.page.index'),

        // the icon that will be shown in the navigation
        // this file needs to exist in img/
        'icon' => $urlGenerator->imagePath('myapp', 'app.svg'),

        // the title of your application. This will be used in the
        // navigation or on the settings page of your app
        'name' => \OC::$server->getL10N('myapp')->t('My App'),
    ];
});

// execute OCA\MyApp\BackgroundJob\Task::run when cron is called
\OC::$server->getJobList()->add('OCA\MyApp\BackgroundJob\Task');

// execute OCA\MyApp\Hooks\User::deleteUser before a user is being deleted
\OCP\Util::connectHook('OC_User', 'pre_deleteUser', 'OCA\MyApp\Hooks\User', 'deleteUser');
```

Although it is also possible to include JavaScript or CSS for other apps by placing the **addScript** or **addStyle** functions inside this file, it is strongly discouraged, because the file is loaded on each request (also such requests that do not return HTML, but e.g. json or webdav).

```
<?php

\OCP\Util::addScript('myapp', 'script');  // include js/script.js for every app
\OCP\Util::addStyle('myapp', 'style');  // include css/style.css for every app
```

App Metadata

The `appinfo/info.xml` contains metadata about the app:

```
<?xml version="1.0"?>
<info>
    <id>yourappname</id>
    <name>Your App</name>
    <description>Your App description</description>
    <version>1.0</version>
    <licence>AGPL</licence>
    <author>Your Name</author>
    <namespace>YourAppsNamespace</namespace>

    <types>
```

```
        <filesystem/>
    </types>


    <documentation>
        <user>https://docs.nextcloud.org</user>
        <admin>https://docs.nextcloud.org</admin>
        <developer>https://docs.nextcloud.org</developer>
    </documentation>

    <category>tool</category>

    <website>https://example.org</website>

    <bugs>https://github.com/nextcloud/theapp/issues</bugs>

    <repository type="git">https://github.com/nextcloud/theapp.git</repository>

    <ocsid>1234</ocsid>

    <dependencies>
        <php min-version="5.6" max-version="7.1"/>
        <database>sqlite</database>
        <database>mysql</database>
        <command os="linux">grep</command>
        <command os="windows">notepad.exe</command>
        <lib min-version="1.2">xml</lib>
        <lib max-version="2.0">intl</lib>
        <lib>curl</lib>
        <os>Linux</os>
        <owncloud min-version="6.0.4" max-version="8"/>
    </dependencies>

    <settings>
        <admin-section>OCA\YourAppsNamespace\Settings\AdminSection</admin-section>
        <admin>OCA\YourAppsNamespace\Settings\AdminSettings</admin>
    </settings>

    <!-- deprecated, just for reference -->
    <requiremin>5</requiremin>
    <public>
        <file id="caldav">appinfo/caldav.php</file>
    </public>

    <remote>
        <file id="caldav">appinfo/caldav.php</file>
    </remote>

    <standalone />

    <default_enable />
    <shipped>true</shipped>
    <!-- end deprecated -->
</info>
```

id

Required: This field contains the internal app name, and has to be the same as the folder name of the app. This id needs to be unique in Nextcloud, meaning no other app should have this id.

name

Required: This is the human-readable name/title of the app that will be displayed in the app overview page.

description

Required: This contains the description of the app which will be shown in the app overview page.

version

Contains the version of your app.

licence

Required: The licence of the app. This licence must be compatible with the AGPL and **must not be proprietary**, for instance:

- AGPL 3 (recommended)
- MIT

author

Required: The name of the app author or authors.

namespace

Required if routes.php returns an array. If your app is namespaced like **\OCA\MyApp\Controller\PageController** the required namespace value is **MyApp**. If not given it tries to default to the first letter upper cased app id, e.g. **myapp** would be tried under **Myapp**

types

Nextcloud allows to specify four kind of `types`. Currently supported `types`:

- **prelogin**: apps which need to load on the login page
- **filesystem**: apps which provide filesystem functionality (e.g. files sharing app)
- **authentication**: apps which provide authentication backends
- **logging**: apps which implement a logging system
- **prevent_group_restriction**: apps which can not be enabled for specific groups (e.g. notifications app).

Note: Due to technical reasons apps of any type listed above can not be enabled for specific groups only.

documentation

Link to 'admin', 'user', 'developer' documentation

website

Link to project web page

repository

Link to the version control repo

bugs

Link to the bug tracker

category

Category on the app store. Can be one of the following:

- auth
- customization
- files
- games
- integration
- monitoring
- multimedia
- office
- organization
- social
- tools

ocsid

The app's id on the app store, e.g.: https://apps.owncloud.com/content/show.php/QOwnNotes?content=168497 would have the ocsid **168497**. If given helps users to install and update the same app from the app store

Dependencies

All tags within the dependencies tag define a set of requirements which have to be fulfilled in order to operate properly. As soon as one of these requirements is not met the app cannot be installed.

php

Defines the minimum and the maximum version of PHP which is required to run this app.

database

Each supported database has to be listed in here. Valid values are sqlite, mysql, pgsql, oci and mssql. In the future it will be possible to specify versions here as well. In case no database is specified it is assumed that all databases are supported.

command

Defines a command line tool to be available. With the attribute 'os' the required operating system for this tool can be specified. Valid values for the 'os' attribute are as returned by the PHP function php_uname.

lib

Defines a required PHP extension with required minimum and/or maximum version. The names for the libraries have to match the result as returned by the PHP function get_loaded_extensions. The explicit version of an extension is read from phpversion - with some exception as to be read up in the code base

os

Defines the required target operating system the app can run on. Valid values are as returned by the PHP function php_uname.

owncloud

Required: Defines minimum and maximum versions of the Nextcloud core. In case undefined the values will be taken from the tag *requiremin*.

Note: Currently this tag is also used to check for the nextcloud version number. Thereby the following "translation" is made:

- ownCloud 9.0 matches Nextcloud 9
- ownCloud 9.1 matches Nextcloud 10
- ownCloud 9.2 matches Nextcloud 11

---

settings

When your app has admin settings, this is the place to register the corresponding classes.

admin-section

In case the app needs to register a new section on the admin settings page, it needs to implement the OCPSettingsISection interface. The implementing class needs to be specified here.

admin

In case the app has its own admin related settings, it needs to implement the OCPSettingsISettings interface. The implementing class needs to be specified here.

Deprecated

The following sections are just listed for reference and should not be used because

requiremin

Deprecated in favor of the **<dependencies>** tag.

public

Used to provide a public interface (requires no login) for the app. The id is appended to the URL **/index.php/public**. Example with id set to 'calendar':

```
/index.php/public/calendar
```

Also take a look at External API.

remote

Same as public but requires login. The id is appended to the URL **/index.php/remote**. Example with id set to 'calendar':

```
/index.php/remote/calendar
```

Also take a look at External API.

standalone

Can be set to true to indicate that this app is a webapp. This can be used to tell GNOME Web for instance to treat this like a native application.

default_enable

Core apps only: Used to tell Nextcloud to enable them after the installation.

shipped

Core apps only: Used to tell Nextcloud that the app is in the standard release.

Please note that if this attribute is set to *FALSE* or not set at all, every time you disable the application, all the files of the application itself will be *REMOVED* from the server!

Classloader

The classloader is provided by Nextcloud and loads all your classes automatically. The only thing left to include by yourself are 3rdparty libraries. Those should be loaded in `lib/AppInfo/Application.php`.

New in version 10.

PSR-4 Autoloading

Since Nextcloud 10 there is a PSR-4 autoloader in place. The namespace **\OCA\MyApp** is mapped to `/apps/myapp/lib/`. Afterwards normal PSR-4 rules apply, so a folder is a namespace section in the same casing and the class name matches the file name.

If your appid can not be turned into the namespace by uppercasing the first character, you can specify it in your **appinfo/info.xml** by providing a field called **namespace**. The required namespace is the one which comes after the top level namespace **OCA**, e.g.: for **OCA\MyBeautifulApp\Some\OtherClass** the needed namespace would be **MyBeautifulApp** and would be added to the info.xml in the following way:

```
<?xml version="1.0"?>
<info>
   <namespace>MyBeautifulApp</namespace>
   <!-- other options here ... -->
</info>
```

A second PSR-4 root is available when running tests. **\OCA\MyApp\Tests** is thereby mapped to `/apps/myapp/tests/`.

Legacy Autoloading

The legacy classloader, deprecated since 10, is still in place and works like this:

- Take the full qualifier of a class:

```
\OCA\MyApp\Controller\PageController
```

- If it starts with \OCA include file from the apps directory
- Cut off \OCA:

```
\MyApp\Controller\PageController
```

- Convert all charactes to lowercase:

```
\myapp\controller\pagecontroller
```

- Replace \ with /:

```
/myapp/controller/pagecontroller
```

- Append .php:

```
/myapp/controller/pagecontroller.php
```

- Prepend /apps because of the **OCA** namespace and include the file:

```
require_once '/apps/myapp/controller/pagecontroller.php';
```

In other words: In order for the PageController class to be autoloaded, the class **\OCA\MyApp\Controller\PageController** needs to be stored in the `/apps/myapp/controller/pagecontroller.php`

Request lifecycle

A typical HTTP request consists of the following:

- **An URL**: e.g. /index.php/apps/myapp/something
- **Request Parameters**: e.g. ?something=true&name=tom
- **A Method**: e.g. GET
- **Request headers**: e.g. Accept: application/json

The following sections will present an overview over how that request is being processed to provide an in depth view over how Nextcloud works. If you are not interested in the internals or don't want to execute anything before and after your controller, feel free to skip this section and continue directly with defining your app's routes.

Front controller

In the beginning, all requests are sent to Nextcloud's `index.php` which in turn executes `lib/base.php`. This file inspects the HTTP headers and abstracts away differences between different Web servers and initializes the basic classes. Afterwards the basic apps are being loaded in the following order:

- Authentication backends
- Filesystem
- Logging

The type of the app is determined by inspecting the app's configuration file (`appinfo/info.xml`). Loading apps means that the main file (`appinfo/app.php`) of each installed app is being loaded and executed. That means that if you want to execute code before a specific app is being run, you can place code in your app's Navigation and Pre-App configuration file.

Afterwards the following steps are performed:

- Try to authenticate the user
- Load and execute all the remaining apps' Navigation and Pre-App configuration files
- Load and run all the routes in the apps' `appinfo/routes.php`
- Execute the router

Router

The router parses the app's routing files (`appinfo/routes.php`), inspects the request's **method** and **url**, queries the controller from the Container and then passes control to the dispatcher. The dispatcher is responsible for running the hooks (called Middleware) before and after the controller, executing the controller method and rendering the output.

Middleware

A Middleware is a convenient way to execute common tasks such as custom authentication before or after a controller method is being run. You can execute code at the following locations:

- before the call of the controller method
- after the call of the controller method
- after an exception is thrown (also if it is thrown from a middleware, e.g. if an authentication fails)
- before the output is rendered

Container

The Container is the place where you define all of your classes and in particular all of your controllers. The container is responsible for assembling all of your objects (instantiating your classes) that should only have one single instance without relying on globals or singletons. If you want to know more about why you should use it and what the benefits are, read up on the topic in Container.

Controller

The controller contains the code that you actually want to run after a request has come in. Think of it like a callback that is executed if everything before went fine.

The controller returns a response which is then run through the middleware again (afterController and beforeOutput hooks are being run), HTTP headers are being set and the response's render method is being called and printed.

Routing

Routes map an URL and a method to a controller method. Routes are defined inside `appinfo/routes.php` by passing a configuration array to the registerRoutes method. An example route would look like this:

```
<?php
namespace OCA\MyApp\AppInfo;

$application = new Application();
$application->registerRoutes($this, array(
    'routes' => array(
        array('name' => 'page#index', 'url' => '/', 'verb' => 'GET'),
    )
));
```

The route array contains the following parts:

- **url**: The url that is matched after */index.php/apps/myapp*

- **name**: The controller and the method to call; *page#index* is being mapped to *PageController->index()*, *articles_api#drop_latest* would be mapped to *ArticlesApiController->dropLatest()*. The controller that matches the page#index name would have to be registered in the following way inside `appinfo/application.php`:

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Controller\PageController;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('PageController', function($c) {
            return new PageController(
                $c->query('AppName'),
                $c->query('Request')
            );
        });
    }

}
```

- **method** (Optional, defaults to GET): The HTTP method that should be matched, (e.g. GET, POST, PUT, DELETE, HEAD, OPTIONS, PATCH)
- **requirements** (Optional): lets you match and extract URLs that have slashes in them (see **Matching suburls**)
- **postfix** (Optional): lets you define a route id postfix. Since each route name will be transformed to a route id (**page#method** -> **myapp.page.method**) and the route id can only exist once you can use the postfix option to alter the route id creation by adding a string to the route id e.g.: **'name' => 'page#method', 'postfix' => 'test'** will yield the route id **myapp.page.methodtest**. This makes it possible to add more than one route/url for a controller method
- **defaults** (Optional): If this setting is given, a default value will be assumed for each url parameter which is not present. The default values are passed in as a key => value par array

Extracting values from the URL

It is possible to extract values from the URL to allow RESTful URL design. To extract a value, you have to wrap it inside curly braces:

```
<?php

// Request: GET /index.php/apps/myapp/authors/3

// appinfo/routes.php
array('name' => 'author#show', 'url' => '/authors/{id}', 'verb' => 'GET'),
```

```
// controller/authorcontroller.php
class AuthorController {

    public function show($id) {
        // $id is '3'
    }

}
```

The identifier used inside the route is being passed into controller method by reflecting the method parameters. So basically if you want to get the value **{id}** in your method, you need to add **$id** to your method parameters.

Matching suburls

Sometimes its needed to match more than one URL fragment. An example would be to match a request for all URLs that start with **OPTIONS /index.php/apps/myapp/api**. To do this, use the **requirements** parameter in your route which is an array containing pairs of **'key' => 'regex'**:

```
<?php

// Request: OPTIONS /index.php/apps/myapp/api/my/route

// appinfo/routes.php
array('name' => 'author_api#cors', 'url' => '/api/{path}', 'verb' => 'OPTIONS',
      'requirements' => array('path' => '.+')),

// controller/authorapicontroller.php
class AuthorApiController {

    public function cors($path) {
        // $path will be 'my/route'
    }

}
```

Default values for suburl

Apart from matching requirements, a suburl may also have a default value. Say you want to support pagination (a 'page' parameter) for your **/posts** suburl that displays posts entries list. You may set a default value for the 'page' parameter, that will be used if not already set in the url. Use the **defaults** parameter in your route which is an array containing pairs of **'urlparameter' => 'defaultvalue'**:

```
<?php

// Request: GET /index.php/app/myapp/post

// appinfo/routes.php
array(
    'name'     => 'post#index',
    'url'      => '/post/{page}',
    'verb'     => 'GET',
    'defaults' => array('page' => 1) // this allows same url as /index.php/myapp/post/1
),

// controller/postcontroller.php
```

```
class PostController
{
    public function index($page = 1)
    {
        // $page will be 1
    }
}
```

Registering resources

When dealing with resources, writing routes can become quite repetitive since most of the time routes for the following tasks are needed:

- Get all entries
- Get one entry by id
- Create an entry
- Update an entry
- Delete an entry

To prevent repetition, it's possible to define resources. The following routes:

```
<?php
namespace OCA\MyApp\AppInfo;

$application = new Application();
$application->registerRoutes($this, array(
    'routes' => array(
        array('name' => 'author#index', 'url' => '/authors', 'verb' => 'GET'),
        array('name' => 'author#show', 'url' => '/authors/{id}', 'verb' => 'GET'),
        array('name' => 'author#create', 'url' => '/authors', 'verb' => 'POST'),
        array('name' => 'author#update', 'url' => '/authors/{id}', 'verb' => 'PUT'),
        array('name' => 'author#destroy', 'url' => '/authors/{id}', 'verb' => 'DELETE'),
        // your other routes here
    )
));
```

can be abbreviated by using the **resources** key:

```
<?php
namespace OCA\MyApp\AppInfo;

$application = new Application();
$application->registerRoutes($this, array(
    'resources' => array(
        'author' => array('url' => '/authors')
    ),
    'routes' => array(
        // your other routes here
    )
));
```

Using the URLGenerator

Sometimes its useful to turn a route into a URL to make the code independent from the URL design or to generate an URL for an image in **img/**. For that specific use case, the ServerContainer provides a service that can be used in your container:

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Controller\PageController;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('PageController', function($c) {
            return new PageController(
                $c->query('AppName'),
                $c->query('Request'),

                // inject the URLGenerator into the page controller
                $c->query('ServerContainer')->getURLGenerator()
            );
        });
    }

}
```

Inside the PageController the URL generator can now be used to generate an URL for a redirect:

```
<?php
namespace OCA\MyApp\Controller;

use \OCP\IRequest;
use \OCP\IURLGenerator;
use \OCP\AppFramework\Controller;
use \OCP\AppFramework\Http\RedirectResponse;

class PageController extends Controller {

    private $urlGenerator;

    public function __construct($appName, IRequest $request,
                                IURLGenerator $urlGenerator) {
        parent::__construct($appName, $request);
        $this->urlGenerator = $urlGenerator;
    }

    /**
     * redirect to /apps/news/myapp/authors/3
```

```
     */
    public function redirect() {
        // route name: author_api#do_something
        // route url: /apps/news/myapp/authors/{id}

        // # needs to be replaced with a . due to limitations and prefixed
        // with your app id
        $route = 'myapp.author_api.do_something';
        $parameters = array('id' => 3);

        $url = $this->urlGenerator->linkToRoute($route, $parameters);

        return new RedirectResponse($url);
    }

}
```

URLGenerator is case sensitive, so **appName** must match **exactly** the name you use in configuration. If you use a CamelCase name as *myCamelCaseApp*,

```
<?php
$route = 'myCamelCaseApp.author_api.do_something';
```

Middleware

Middleware is logic that is run before and after each request and is modelled after Django's Middleware system. It offers the following hooks:

- **beforeController**: This is executed before a controller method is being executed. This allows you to plug additional checks or logic before that method, like for instance security checks
- **afterException**: This is being run when either the beforeController method or the controller method itself is throwing an exception. The middleware is asked in reverse order to handle the exception and to return a response. If the middleware can't handle the exception, it throws the exception again
- **afterController**: This is being run after a successful controllermethod call and allows the manipulation of a Response object. The middleware is run in reverse order
- **beforeOutput**: This is being run after the response object has been rendered and allows the manipulation of the outputted text. The middleware is run in reverse order

To generate your own middleware, simply inherit from the Middleware class and overwrite the methods that should be used.

```
<?php

namespace OCA\MyApp\Middleware;

use \OCP\AppFramework\Middleware;


class CensorMiddleware extends Middleware {

    /**
     * this replaces "bad words" with "********" in the output
     */
    public function beforeOutput($controller, $methodName, $output){
```

```
        return str_replace('bad words', '********', $output);
    }

}
```

The middleware can be registered in the Container and added using the **registerMiddleware** method:

```
<?php

namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Middleware\CensorMiddleware;

class MyApp extends App {

    /**
     * Define your dependencies in here
     */
    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Middleware
         */
        $container->registerService('CensorMiddleware', function($c){
            return new CensorMiddleware();
        });

        // executed in the order that it is registered
        $container->registerMiddleware('CensorMiddleware');

    }
}
```

Note: The order is important! The middleware that is registered first gets run first in the **beforeController** method. For all other hooks, the order is being reversed, meaning: if a middleware is registered first, it gets run last.

Parsing annotations

Sometimes its useful to conditionally execute code before or after a controller method. This can be done by defining custom annotations. An example would be to add a custom authentication method or simply add an additional header to the response. To access the parsed annotations, inject the **ControllerMethodReflector** class:

```
<?php

namespace OCA\MyApp\Middleware;

use \OCP\AppFramework\Middleware;
use \OCP\AppFramework\Utility\ControllerMethodReflector;
use \OCP\IRequest;
```

```
class HeaderMiddleware extends Middleware {

  private $reflector;

  public function __construct(ControllerMethodReflector $reflector) {
      $this->reflector = $reflector;
  }

  /**
   * Add custom header if @MyHeader is used
   */
  public function afterController($controller, $methodName, IResponse $response){
      if($this->reflector->hasAnnotation('MyHeader')) {
          $response->addHeader('My-Header', 3);
      }
      return $response;
  }

}
```

Now adjust the container to inject the reflector:

```
<?php

namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Middleware\HeaderMiddleware;

class MyApp extends App {

    /**
     * Define your dependencies in here
     */
    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Middleware
         */
        $container->registerService('HeaderMiddleware', function($c){
            return new HeaderMiddleware($c->query('ControllerMethodReflector'));
        });

        // executed in the order that it is registered
        $container->registerMiddleware('HeaderMiddleware');
    }

}
```

Note: An annotation always starts with an uppercase letter

Container

The App Framework assembles the application by using a container based on the software pattern Dependency Injection. This makes the code easier to test and thus easier to maintain.

If you are unfamiliar with this pattern, watch the following videos:

- Dependency Injection and the art of Services and Containers Tutorial
- Google Clean Code Talks

Dependency Injection

Dependency Injection sounds pretty complicated but it just means: Don't put new dependencies in your constructor or methods but pass them in. So this:

```
<?php

// without dependency injection
class AuthorMapper {

  private $db;

  public function __construct() {
    $this->db = new Db();
  }

}
```

would turn into this by using Dependency Injection:

```
<?php

// with dependency injection
class AuthorMapper {

  private $db;

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

}
```

Using a container

Passing dependencies into the constructor rather than instantiating them in the constructor has the following drawback: Every line in the source code where **new AuthorMapper** is being used has to be changed, once a new constructor argument is being added to it.

The solution for this particular problem is to limit the **new AuthorMapper** to one file, the container. The container contains all the factories for creating these objects and is configured in `lib/AppInfo/Application.php`.

To add the app's classes simply open the `lib/AppInfo/Application.php` and use the **registerService** method on the container object:

```php
<?php

namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Controller\AuthorController;
use \OCA\MyApp\Service\AuthorService;
use \OCA\MyApp\Db\AuthorMapper;

class Application extends App {


  /**
   * Define your dependencies in here
   */
  public function __construct(array $urlParams=array()){
    parent::__construct('myapp', $urlParams);

    $container = $this->getContainer();

    /**
     * Controllers
     */
    $container->registerService('AuthorController', function($c){
      return new AuthorController(
        $c->query('AppName'),
        $c->query('Request'),
        $c->query('AuthorService')
      );
    });

    /**
     * Services
     */
    $container->registerService('AuthorService', function($c){
      return new AuthorService(
        $c->query('AuthorMapper')
      );
    });

    /**
     * Mappers
     */
    $container->registerService('AuthorMapper', function($c){
      return new AuthorMapper(
        $c->query('ServerContainer')->getDb()
      );
    });
  }
}
```

How the container works

The container works in the following way:

- A request comes in and is matched against a route (for the AuthorController in this case)

- The matched route queries **AuthorController** service from the container:

```
return new AuthorController(
  $c->query('AppName'),
  $c->query('Request'),
  $c->query('AuthorService')
);
```

- The **AppName** is queried and returned from the baseclass
- The **Request** is queried and returned from the server container
- **AuthorService** is queried:

```
$container->registerService('AuthorService', function($c){
  return new AuthorService(
    $c->query('AuthorMapper')
  );
});
```

- **AuthorMapper** is queried:

```
$container->registerService('AuthorMappers', function($c){
  return new AuthorService(
    $c->query('ServerContainer')->getDb()
  );
});
```

- The **database connection** is returned from the server container
- Now **AuthorMapper** has all of its dependencies and the object is returned
- **AuthorService** gets the **AuthorMapper** and returns the object
- **AuthorController** gets the **AuthorService** and finally the controller can be instantiated and the object is returned

So basically the container is used as a giant factory to build all the classes that are needed for the application. Because it centralizes all the creation of objects (the **new Class()** lines), it is very easy to add new constructor parameters without breaking existing code: only the **__construct** method and the container line where the **new** is being called need to be changed.

Use automatic dependency assembly (recommended)

In Nextcloud it is possible to omit the **lib/AppInfo/Application.php** and use automatic dependency assembly instead.

How does automatic assembly work

Automatic assembly creates new instances of classes just by looking at the class name and its constructor parameters. For each constructor parameter the type or the variable name is used to query the container, e.g.:

- **SomeType $type** will use **$container->query('SomeType')**
- **$variable** will use **$container->query('variable')**

If all constructor parameters are resolved, the class will be created, saved as a service and returned.

So basically the following is now possible:

```php
<?php
namespace OCA\MyApp;

class MyTestClass {}

class MyTestClass2 {
    public $class;
    public $appName;

    public function __construct(MyTestClass $class, $AppName) {
        $this->class = $class;
        $this->appName = $AppName;
    }
}

$app = new \OCP\AppFramework\App('myapp');

$class2 = $app->getContainer()->query('OCA\MyApp\MyTestClass2');

$class2 instanceof MyTestClass2;  // true
$class2->class instanceof MyTestClass;  // true
$class2->appName === 'appname';  // true
$class2 === $app->getContainer()->query('OCA\MyApp\MyTestClass2');  // true
```

Note: $AppName is resolved because the container registered a parameter under the key 'AppName' which will return the app id. The lookup is case sensitive so while $AppName will work correctly, using $appName as a constructor parameter will fail.

How does it affect the request lifecycle

- A request comes in
- All apps' **routes.php** files are loaded
  - If a **routes.php** file returns an array, and an **appname/lib/AppInfo/Application.php** exists, include it, create a new instance of **\OCA\AppName\AppInfo\Application.php** and register the routes on it. That way a container can be used while still benefitting from the new routes behavior
  - If a **routes.php** file returns an array, but there is no **appname/lib/AppInfo/Application.php**, create a new \OCP\AppFramework\App instance with the app id and register the routes on it
- A request is matched for the route, e.g. with the name **page#index**
- The appropriate container is being queried for the entry PageController (to keep backwards compatibility)
- If the entry does not exist, the container is queried for OCA\AppName\Controller\PageController and if no entry exists, the container tries to create the class by using reflection on its constructor parameters

How does this affect controllers

The only thing that needs to be done to add a route and a controller method is now:

myapp/appinfo/routes.php

```
<?php
return ['routes' => [
    ['name' => 'page#index', 'url' => '/', 'verb' => 'GET'],
]];
```

myapp/appinfo/lib/Controller/PageController.php

```
<?php
namespace OCA\MyApp\Controller;

class PageController {
    public function __construct($AppName, \OCP\IRequest $request) {
        parent::__construct($AppName, $request);
    }

    public function index() {
        // your code here
    }
}
```

There is no need to wire up anything in **lib/AppInfo/Application.php**. Everything will be done automatically.

How to deal with interface and primitive type parameters

Interfaces and primitive types can not be instantiated, so the container can not automatically assemble them. The actual implementation needs to be wired up in the container:

```
<?php

namespace OCA\MyApp\AppInfo;

class Application extends \OCP\AppFramework\App {

    /**
     * Define your dependencies in here
     */
    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        // AuthorMapper requires a location as string called $TableName
        $container->registerParameter('TableName', 'my_app_table');

        // the interface is called IAuthorMapper and AuthorMapper implements it
        $container->registerService('OCA\MyApp\Db\IAuthorMapper', function ($c) {
            return $c->query('OCA\MyApp\Db\AuthorMapper');
        });
    }

}
```

Predefined core services

The following parameter names and type hints can be used to inject core services instead of using **$container->getServer()->getServiceX()**

Parameters:

- **AppName**: The app id
- **WebRoot**: The path to the Nextcloud installation
- **UserId**: The id of the current user

Types:

- **OCP\IAppConfig**
- **OCP\IAppManager**
- **OCP\IAvatarManager**
- **OCP\Activity\IManager**
- **OCP\ICache**
- **OCP\ICacheFactory**
- **OCP\IConfig**
- **OCP\AppFramework\Utility\IControllerMethodReflector**
- **OCP\Contacts\IManager**
- **OCP\IDateTimeZone**
- **OCP\IDBConnection**
- **OCP\Diagnostics\IEventLogger**
- **OCP\Diagnostics\IQueryLogger**
- **OCP\Files\Config\IMountProviderCollection**
- **OCP\Files\IRootFolder**
- **OCP\IGroupManager**
- **OCP\IL10N**
- **OCP\ILogger**
- **OCP\BackgroundJob\IJobList**
- **OCP\INavigationManager**
- **OCP\IPreview**
- **OCP\IRequest**
- **OCP\AppFramework\Utility\ITimeFactory**
- **OCP\ITagManager**
- **OCP\ITempManager**
- **OCP\Route\IRouter**
- **OCP\ISearch**
- **OCP\ISearch**
- **OCP\Security\ICrypto**
- **OCP\Security\IHasher**
- **OCP\Security\ISecureRandom**

- **OCP\IURLGenerator**
- **OCP\IUserManager**
- **OCP\IUserSession**

How to enable it

To make use of this new feature, the following things have to be done:

- **appinfo/info.xml** requires to provide another field called **namespace** where the namespace of the app is defined. The required namespace is the one which comes after the top level namespace **OCA**, e.g.: for **OCA\MyBeautifulApp\Some\OtherClass** the needed namespace would be **MyBeautifulApp** and would be added to the info.xml in the following way:

```
<?xml version="1.0"?>
<info>
   <namespace>MyBeautifulApp</namespace>
   <!-- other options here ... -->
</info>
```

- **appinfo/routes.php**: Instead of creating a new Application class instance, simply return the routes array like:

```
<?php
return ['routes' => [
    ['name' => 'page#index', 'url' => '/', 'verb' => 'GET'],
]];
```

Note: A namespace tag is required because you can not deduce the namespace from the app id

Which classes should be added

In general all of the app's controllers need to be registered inside the container. Then the following question is: What goes into the constructor of the controller? Pass everything into the controller constructor that matches one of the following criteria:

- It does I/O (database, write/read to files)
- It is a global (e.g. $_POST, etc. This is in the request class by the way)
- The output does not depend on the input variables (also called impure function), e.g. time, random number generator
- It is a service, basically it would make sense to swap it out for a different object

What not to inject:

- It is pure data and has methods that only act upon it (arrays, data objects)
- It is a pure function

Controllers

Controllers are used to connect routes with app logic. Think of it as callbacks that are executed once a request has come in. Controllers are defined inside the **lib/Controller/** directory.

To create a controller, simply extend the Controller class and create a method that should be executed on a request:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class AuthorController extends Controller {

    public function index() {

    }

}
```

Connecting a controller and a route

To connect a controller and a route the controller has to be registered in the Container like this:

```
<?php
namespace OCA\MyApp\AppInfo;

use OCP\AppFramework\App;

use OCA\MyApp\Controller\AuthorApiController;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('AuthorApiController', function($c) {
            return new AuthorApiController(
                $c->query('AppName'),
                $c->query('Request')
            );
        });
    }
}
```

Every controller needs the app name and the request object passed into their parent constructor, which can easily be injected like shown in the example code above. The important part is not the class name but rather the string which is passed in as the first parameter of the **registerService** method.

The other part is the route name. An example route name would look like this:

```
author_api#some_method
```

This name is processed in the following way:

- Remove the underscore and uppercase the next character:

```
authorApi#someMethod
```

- Split at the # and uppercase the first letter of the left part:

```
AuthorApi
someMethod
```

- Append Controller to the first part:

```
AuthorApiController
someMethod
```

- Now retrieve the service listed under **AuthorApiController** from the container, look up the parameters of the **someMethod** method in the request, cast them if there are PHPDoc type annotations and execute the **someMethod** method on the controller with those parameters.

Getting request parameters

Parameters can be passed in many ways:

- Extracted from the URL using curly braces like **{key}** inside the URL (see Routing)
- Appended to the URL as a GET request (e.g. ?something=true)
- application/x-www-form-urlencoded from a form or jQuery
- application/json from a POST, PATCH or PUT request

All those parameters can easily be accessed by adding them to the controller method:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class PageController extends Controller {

    // this method will be executed with the id and name parameter taken
    // from the request
    public function doSomething($id, $name) {

    }

}
```

It is also possible to set default parameter values by using PHP default method values so common values can be omitted:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class PageController extends Controller {

    /**
     * @param int $id
     */
    public function doSomething($id, $name='john', $job='author') {
        // GET ?id=3&job=killer
```

```
        // $id = 3
        // $name = 'john'
        // $job = 'killer'
    }

}
```

Casting parameters

URL, GET and application/x-www-form-urlencoded have the problem that every parameter is a string, meaning that:

```
?doMore=false
```

would be passed in as the string *'false'* which is not what one would expect. To cast these to the correct types, simply add PHPDoc in the form of:

```
@param type $name
```

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class PageController extends Controller {

    /**
     * @param int $id
     * @param bool $doMore
     * @param float $value
     */
    public function doSomething($id, $doMore, $value) {
        // GET /index.php/apps/myapp?id=3&doMore=false&value=3.5
        // => $id = 3
        //    $doMore = false
        //    $value = 3.5
    }

}
```

The following types will be cast:

- **bool** or **boolean**
- **float**
- **int** or **integer**

JSON parameters

It is possible to pass JSON using a POST, PUT or PATCH request. To do that the **Content-Type** header has to be set to **application/json**. The JSON is being parsed as an array and the first level keys will be used to pass in the arguments, e.g.:

```
POST /index.php/apps/myapp/authors
Content-Type: application/json
{
    "name": "test",
```

```
    "number": 3,
    "publisher": true,
    "customFields": {
        "mail": "test@example.com",
        "address": "Somewhere"
    }
}
```

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class PageController extends Controller {

    public function create($name, $number, $publisher, $customFields) {
        // $name = 'test'
        // $number = 3
        // $publisher = true
        // $customFields = array("mail" => "test@example.com", "address" => "Somewhere")
    }

}
```

Reading headers, files, cookies and environment variables

Headers, files, cookies and environment variables can be accessed directly from the request object:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\IRequest;

class PageController extends Controller {

    public function someMethod() {
        $type = $this->request->getHeader('Content-Type');  // $_SERVER['HTTP_CONTENT_TYPE']
        $cookie = $this->request->getCookie('myCookie');  // $_COOKIES['myCookie']
        $file = $this->request->getUploadedFile('myfile');  // $_FILES['myfile']
        $env = $this->request->getEnv('SOME_VAR');  // $_ENV['SOME_VAR']
    }

}
```

Why should those values be accessed from the request object and not from the global array like $_FILES? Simple: because it's bad practice and will make testing harder.

Reading and writing session variables

To set, get or modify session variables, the ISession object has to be injected into the controller.

Then session variables can be accessed like this:

Note: The session is closed automatically for writing, unless you add the @UseSession annotation!

```php
<?php
namespace OCA\MyApp\Controller;

use OCP\ISession;
use OCP\IRequest;
use OCP\AppFramework\Controller;

class PageController extends Controller {

    private $session;

    public function __construct($AppName, IRequest $request, ISession $session) {
        parent::__construct($AppName, $request);
        $this->session = $session;
    }

    /**
     * The following annotation is only needed for writing session values
     * @UseSession
     */
    public function writeASessionVariable() {
        // read a session variable
        $value = $this->session['value'];

        // write a session variable
        $this->session['value'] = 'new value';
    }

}
```

Setting cookies

Cookies can be set or modified directly on the response class:

```php
<?php
namespace OCA\MyApp\Controller;

use DateTime;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\TemplateResponse;
use OCP\IRequest;

class BakeryController extends Controller {

    /**
     * Adds a cookie "foo" with value "bar" that expires after user closes the browser
     * Adds a cookie "bar" with value "foo" that expires 2015-01-01
     */
    public function addCookie() {
        $response = new TemplateResponse(...);
        $response->addCookie('foo', 'bar');
        $response->addCookie('bar', 'foo', new DateTime('2015-01-01 00:00'));
        return $response;
    }

    /**
```

```
        * Invalidates the cookie "foo"
        * Invalidates the cookie "bar" and "bazinga"
        */
     public function invalidateCookie() {
         $response = new TemplateResponse(...);
         $response->invalidateCookie('foo');
         $response->invalidateCookies(array('bar', 'bazinga'));
         return $response;
     }
}
```

Responses

Similar to how every controller receives a request object, every controller method has to to return a Response. This can be in the form of a Response subclass or in the form of a value that can be handled by a registered responder.

JSON

Returning JSON is simple, just pass an array to a JSONResponse:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\JSONResponse;

class PageController extends Controller {

    public function returnJSON() {
        $params = array('test' => 'hi');
        return new JSONResponse($params);
    }

}
```

Because returning JSON is such an common task, there's even a shorter way to do this:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;

class PageController extends Controller {

    public function returnJSON() {
        return array('test' => 'hi');
    }

}
```

Why does this work? Because the dispatcher sees that the controller did not return a subclass of a Response and asks the controller to turn the value into a Response. That's where responders come in.

Responders

Responders are short functions that take a value and return a response. They are used to return different kinds of responses based on a **format** parameter which is supplied by the client. Think of an API that is able to return both XML and JSON depending on if you call the URL with:

```
?format=xml
```

or:

```
?format=json
```

The appropriate responder is being chosen by the following criteria:

- First the dispatcher checks the Request if there is a **format** parameter, e.g.:

```
    ?format=xml
```

  or:

```
    /index.php/apps/myapp/authors.{format}
```

- If there is none, take the **Accept** header, use the first mimetype and cut off *application/*. In the following example the format would be *xml*:

```
    Accept: application/xml, application/json
```

- If there is no Accept header or the responder does not exist, format defaults to **json**.

By default there is only a responder for JSON but more can be added easily:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\DataResponse;

class PageController extends Controller {

    public function returnHi() {

        // XMLResponse has to be implemented
        $this->registerResponder('xml', function($value) {
            if ($value instanceof DataResponse) {
                return new XMLResponse(
                    $value->getData(),
                    $value->getStatus(),
                    $value->getHeaders()
                );
            } else {
                return new XMLResponse($value);
            }
        });

        return array('test' => 'hi');
    }

}
```

Note: The above example would only return XML if the **format** parameter was *xml*. If you want to return an

XMLResponse regardless of the format parameter, extend the Response class and return a new instance of it from the controller method instead.

Because returning values works fine in case of a success but not in case of failure that requires a custom HTTP error code, you can always wrap the value in a **DataResponse**. This works for both normal responses and error responses.

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\Http\Http;

class PageController extends Controller {

    public function returnHi() {
        try {
            return new DataResponse(calculate_hi());
        } catch (\Exception $ex) {
            return new DataResponse(array('msg' => 'not found!'), Http::STATUS_NOT_FOUND);
        }
    }

}
```

Templates

A template can be rendered by returning a TemplateResponse. A TemplateResponse takes the following parameters:

- **appName**: tells the template engine in which app the template should be located
- **templateName**: the name of the template inside the template/ folder without the .php extension
- **parameters**: optional array parameters that are available in the template through $_, e.g.:

```
array('key' => 'something')
```

  can be accessed through:

```
$_['key']
```

- **renderAs**: defaults to *user*, tells Nextcloud if it should include it in the web interface, or in case *blank* is passed solely render the template

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\TemplateResponse;

class PageController extends Controller {

    public function index() {
        $templateName = 'main';  // will use templates/main.php
        $parameters = array('key' => 'hi');
        return new TemplateResponse($this->appName, $templateName, $parameters);
    }

}
```

Redirects

A redirect can be achieved by returning a RedirectResponse:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\RedirectResponse;

class PageController extends Controller {

    public function toGoogle() {
        return new RedirectResponse('https://google.com');
    }

}
```

Downloads

A file download can be triggered by returning a DownloadResponse:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\DownloadResponse;

class PageController extends Controller {

    public function downloadXMLFile() {
        $path = '/some/path/to/file.xml';
        $contentType = 'application/xml';

        return new DownloadResponse($path, $contentType);
    }

}
```

Creating custom responses

If no premade Response fits the needed usecase, its possible to extend the Response baseclass and custom Response. The only thing that needs to be implemented is the **render** method which returns the result as string.

Creating a custom XMLResponse class could look like this:

```
<?php
namespace OCA\MyApp\Http;

use OCP\AppFramework\Http\Response;

class XMLResponse extends Response {
```

```
    private $xml;

    public function __construct(array $xml) {
        $this->addHeader('Content-Type', 'application/xml');
        $this->xml = $xml;
    }

    public function render() {
        $root = new SimpleXMLElement('<root/>');
        array_walk_recursive($this->xml, array ($root, 'addChild'));
        return $xml->asXML();
    }

}
```

Streamed and lazily rendered responses

By default all responses are rendered at once and sent as a string through middleware. In certain cases this is not a desirable behavior, for instance if you want to stream a file in order to save memory. To do that use the now available **OCP\AppFramework\Http\StreamResponse** class:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\StreamResponse;

class PageController extends Controller {

    public function downloadXMLFile() {
        return new StreamResponse('/some/path/to/file.xml');
    }

}
```

If you want to use a custom, lazily rendered response simply implement the interface **OCP\AppFramework\Http\ICallbackResponse** for your response:

```
<?php
namespace OCA\MyApp\Http;

use OCP\AppFramework\Http\Response;
use OCP\AppFramework\Http\ICallbackResponse;

class LazyResponse extends Response implements ICallbackResponse {

    public function callback(IOutput $output) {
        // custom code in here
    }

}
```

Note: Because this code is rendered after several usually built in helpers, you need to take care of errors and proper HTTP caching by yourself.

Modifying the Content Security Policy

By default Nextcloud disables all resources which are not served on the same domain, forbids cross domain requests and disables inline CSS and JavaScript by setting a Content Security Policy. However if an app relies on thirdparty media or other features which are forbidden by the current policy the policy can be relaxed.

Note: Double check your content and edge cases before you relax the policy! Also read the documentation provided by MDN

To relax the policy pass an instance of the ContentSecurityPolicy class to your response. The methods on the class can be chained.

The following methods turn off security features by passing in **true** as the **$isAllowed** parameter

- **allowInlineScript** (bool $isAllowed)
- **allowInlineStyle** (bool $isAllowed)
- **allowEvalScript** (bool $isAllowed)

The following methods whitelist domains by passing in a domain or * for any domain:

- **addAllowedScriptDomain** (string $domain)
- **addAllowedStyleDomain** (string $domain)
- **addAllowedFontDomain** (string $domain)
- **addAllowedImageDomain** (string $domain)
- **addAllowedConnectDomain** (string $domain)
- **addAllowedMediaDomain** (string $domain)
- **addAllowedObjectDomain** (string $domain)
- **addAllowedFrameDomain** (string $domain)
- **addAllowedChildSrcDomain** (string $domain)

The following policy for instance allows images, audio and videos from other domains:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http\TemplateResponse;
use OCP\AppFramework\Http\ContentSecurityPolicy;

class PageController extends Controller {

    public function index() {
        $response = new TemplateResponse('myapp', 'main');
        $csp = new ContentSecurityPolicy();
        $csp->addAllowedImageDomain('*');
            ->addAllowedMediaDomain('*');
        $response->setContentSecurityPolicy($csp);
    }

}
```

OCS

Note: This is purely for compatibility reasons. If you are planning to offer an external API, go for a RESTful API instead.

In order to ease migration from OCS API routes to the App Framework, an additional controller and response have been added. To migrate your API you can use the **OCP\AppFramework\OCSController** baseclass and return your data in the form of a DataResponse in the following way:

```
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Http\DataResponse;
use OCP\AppFramework\OCSController;

class ShareController extends OCSController {

    /**
     * @NoAdminRequired
     * @NoCSRFRequired
     * @PublicPage
     * @CORS
     */
    public function getShares() {
        return new DataResponse([
            //Your data here
        ]);
    }

}
```

The format parameter works out of the box, no intervention is required.

In order to make routing work for OCS routes you need to add a seperate 'ocs' entry to the routing table of your app. Inside these are normal routes.

```
<?php

return [
     'ocs' => [
         [
             'name' => 'Share#getShares',
             'url' => '/api/v1/shares',
             'verb' => 'GET',
         ],
     ],
];
```

Now your method will be reachable via `<server>/ocs/v2.php/apps/<APPNAME>/api/v1/shares`

Handling errors

Sometimes a request should fail, for instance if an author with id 1 is requested but does not exist. In that case use an appropriate HTTP error code to signal the client that an error occurred.

Each response subclass has access to the **setStatus** method which lets you set an HTTP status code. To return a JSONResponse signaling that the author with id 1 has not been found, use the following code:

```php
<?php
namespace OCA\MyApp\Controller;

use OCP\AppFramework\Controller;
use OCP\AppFramework\Http;
use OCP\AppFramework\Http\JSONResponse;

class AuthorController extends Controller {

    public function show($id) {
        try {
            // try to get author with $id

        } catch (NotFoundException $ex) {
            return new JSONResponse(array(), Http::STATUS_NOT_FOUND);
        }
    }

}
```

Authentication

By default every controller method enforces the maximum security, which is:

- Ensure that the user is admin
- Ensure that the user is logged in
- Check the CSRF token

Most of the time though it makes sense to also allow normal users to access the page and the PageController->index() method should not check the CSRF token because it has not yet been sent to the client and because of that can't work.

To turn off checks the following *Annotations* can be added before the controller:

- **@NoAdminRequired**: Also users that are not admins can access the page
- **@NoCSRFRequired**: Don't check the CSRF token (use this wisely since you might create a security hole, to understand what it does see Security Guidelines)
- **@PublicPage**: Everyone can access the page without having to log in

A controller method that turns off all checks would look like this:

```php
<?php
namespace OCA\MyApp\Controller;

use OCP\IRequest;
use OCP\AppFramework\Controller;

class PageController extends Controller {

    /**
     * @NoAdminRequired
     * @NoCSRFRequired
     * @PublicPage
     */
    public function freeForAll() {

    }
```

```
}
```

RESTful API

Offering a RESTful API is not different from creating a route and controllers for the web interface. It is recommended though to inherit from ApiController and add **@CORS** annotations to the methods so that web applications will also be able to access the API.

```
<?php
namespace OCA\MyApp\Controller;

use \OCP\AppFramework\ApiController;
use \OCP\IRequest;

class AuthorApiController extends ApiController {

    public function __construct($appName, IRequest $request) {
        parent::__construct($appName, $request);
    }

    /**
     * @CORS
     */
    public function index() {

    }

}
```

CORS also needs a separate URL for the preflighted **OPTIONS** request that can easily be added by adding the following route:

```
<?php
// appinfo/routes.php
array(
    'name' => 'author_api#preflighted_cors',
    'url' => '/api/1.0/{path}',
    'verb' => 'OPTIONS',
    'requirements' => array('path' => '.+')
)
```

Keep in mind that multiple apps will likely depend on the API interface once it is published and they will move at different speeds to react to changes implemented in the API. Therefore it is recommended to version the API in the URL to not break existing apps when backwards incompatible changes are introduced:

```
/index.php/apps/myapp/api/1.0/resource
```

Modifying the CORS headers

By default the following values will be used for the preflighted OPTIONS request:

- **Access-Control-Allow-Methods**: 'PUT, POST, GET, DELETE, PATCH'
- **Access-Control-Allow-Headers**: 'Authorization, Content-Type, Accept'

- **Access-Control-Max-Age**: 1728000

To add an additional method or header or allow less headers, simply pass additional values to the parent constructor:

```
<?php
namespace OCA\MyApp\Controller;

use \OCP\AppFramework\ApiController;
use \OCP\IRequest;

class AuthorApiController extends ApiController {

    public function __construct($appName, IRequest $request) {
        parent::__construct(
            $appName,
            $request,
            'PUT, POST, GET, DELETE, PATCH',
            'Authorization, Content-Type, Accept',
            1728000);
    }

}
```

Templates

Nextcloud provides its own templating system which is basically plain PHP with some additional functions and preset variables. All the parameters which have been passed from the controller are available in an array called **$_[]**, e.g.:

```
array('key' => 'something')
```

can be accessed through:

```
$_['key']
```

Note: To prevent XSS the following PHP **functions for printing are forbidden: echo, print() and <?=**. Instead use the **p()** function for printing your values. Should you require unescaped printing, **double check for XSS** and use: `print_unescaped`.

Printing values is done by using the **p()** function, printing HTML is done by using **print_unescaped()**

`templates/main.php`

```
<?php foreach($_['entries'] as $entry){ ?>
  <p><?php p($entry); ?></p>
<?php
}
```

Including templates

Templates can also include other templates by using the **$this->inc('templateName')** method.

```
<?php print_unescaped($this->inc('sub.inc')); ?>
```

The parent variables will also be available in the included templates, but should you require it, you can also pass new variables to it by using the second optional parameter as array for **$this->inc**.

templates/sub.inc.php

```
<div>I am included, but I can still access the parents variables!</div>
<?php p($_['name']); ?>

<?php print_unescaped($this->inc('other_template', array('variable' => 'value'))); ?>
```

Including CSS and JavaScript

To include CSS or JavaScript use the **style** and **script** functions:

```
<?php
script('myapp', 'script');  // add js/script.js
style('myapp', 'style');  // add css/style.css
```

Including images

To generate links to images use the **image_path** function:

```
<img src="<?php print_unescaped(image_path('myapp', 'app.png')); ?>" />
```

JavaScript

The JavaScript files reside in the **js/** folder and should be included in the template:

```
<?php
// add one file
script('myapp', 'script');  // adds js/script.js

// add multiple files in the same app
script('myapp', array('script', 'navigation'));  //  adds js/script.js js/navigation.js

// add vendor files (also allows the array syntax)
vendor_script('myapp', 'script');  //  adds vendor/script.js
```

If the script file is only needed when the file list is displayed, you should listen to the `OCA\Files::loadAdditionalScripts` event:

```
<?php
$eventDispatcher = \OC::$server->getEventDispatcher();
$eventDispatcher->addListener('OCA\Files::loadAdditionalScripts', function() {
  script('myapp', 'script');  // adds js/script.js
  vendor_script('myapp', 'script');  //  adds vendor/script.js
});
```

Sending the CSRF token

If any other JavaScript request library than jQuery is being used, the requests need to send the CSRF token as an HTTP header named **requesttoken**. The token is available in the global variable **oc_requesttoken**.

For AngularJS the following lines would need to be added:

```
var app = angular.module('MyApp', []).config(['$httpProvider', function($httpProvider) {
    $httpProvider.defaults.headers.common.requesttoken = oc_requesttoken;
}]);
```

Generating URLs

To send requests to Nextcloud the base URL where Nextcloud is currently running is needed. To get the base URL use:

```
var baseUrl = OC.generateUrl('');
```

Full URLs can be generated by using:

```
var authorUrl = OC.generateUrl('/apps/myapp/authors/1');
```

Extending core parts

It is possible to extend components of the core web UI. The following examples should show how this is possible.

Extending the "new" menu in the files app

New in version 9.0.

```
var myFileMenuPlugin = {
    attach: function (menu) {
        menu.addMenuEntry({
            id: 'abc',
            displayName: 'Menu display name',
            templateName: 'templateName.ext',
            iconClass: 'icon-filetype-text',
            fileType: 'file',
            actionHandler: function () {
                console.log('do something here');
            }
        });
    }
};
OC.Plugins.register('OCA.Files.NewFileMenu', myFileMenuPlugin);
```

This will register a new menu entry in the "New" menu of the files app. The method `attach()` is called once the menu is built. This usually happens right after the click on the button.

CSS

The CSS files reside in the **css/** folder and should be included in the template:

```
<?php
// include one file
style('myapp', 'style');  // adds css/style.css

// include multiple files for the same app
style('myapp', array('style', 'navigation'));  // adds css/style.css, css/navigation.css
```

```
// include vendor file (also allows vendor syntax)
vendor_style('myapp', 'style');  // adds vendor/style.css
```

Web Components go into the **component/** folder and can be imported like this:

```
<?php
// include one file
component('myapp', 'tabs');  // adds component/tabs.html

// include multiple files for the same app
component('myapp', array('tabs', 'forms'));  // adds component/tabs.html, component/forms.html
```

Note: Keep in mind that Web Components are still very new and you might need to add polyfills using Polymer

Standard layout

To use the commonly used layout consisting of sidebar navigation and content the **app-navigation** and **app-content** ids can be used:

```
<div id="app">
    <div id="app-navigation">Your navigation</div>
    <div id="app-content">
        <div id="app-content-wrapper">
            Your content in here
        </div>
    </div>
</div>
```

For built in mobile support your content has to be wrapped inside another div with the id **app-content-wrapper**.

Navigation

Nextcloud provides a default CSS navigation layout. If list entries should have 16x16 px icons, the **with-icon** class can be added to the base **ul**. The maximum supported indention level is two, further indentions are not recommended.

```
<div id="app-navigation">
    <ul class="with-icon">
        <li><a href="#">First level entry</a></li>
        <li>
            <a href="#">First level container</a>
            <ul>
                <li><a href="#">Second level entry</a></li>
                <li><a href="#">Second level entry</a></li>
            </ul>
        </li>
    </ul>
</div>
```

Folders

Folders are like normal entries and are only supported for the first level. In contrast to normal entries, the links which show the title of the folder need to have the **icon-folder** css class.

If the folder should be collapsible, the **collapsible** class and a button with the class **collapse** are needed. After adding the collapsible class the folder's child entries can be toggled by adding the **open** class to the list element:

```
<div id="app-navigation">
    <ul class="with-icon">
        <li><a href="#">First level entry</a></li>
        <li class="collapsible open">
            <button class="collapse"></button>
            <a href="#" class="icon-folder svg">Folder name</a>
            <ul>
                <li><a href="#">Folder contents</a></li>
                <li><a href="#">Folder contents</a></li>
            </ul>
        </li>
    </ul>
</div>
```

Drag and drop

The class which should be applied to a first level element (**li**) that hosts or can host a second level is **drag-and-drop**. This will cause the hovered entry to slide down giving a visual hint that it can accept the dragged element. In case of jQuery UI's droppable feature, the **hoverClass** option should be set to the **drag-and-drop** class.

```
<div id="app-navigation">
    <ul class="with-icon">
        <li><a href="#">First level entry</a></li>
        <li class="drag-and-drop">
            <a href="#" class="icon-folder svg">Folder name</a>
            <ul>
                <li><a href="#">Folder contents</a></li>
                <li><a href="#">Folder contents</a></li>
            </ul>
        </li>
    </ul>
</div>
```

Menus

New in version 8.

To add actions that affect the current list element you can add a menu for second and/or first level elements by adding the button and menu inside the corresponding **li** element and adding the **with-menu** css class:

```
<div id="app-navigation">
    <ul>
        <li class="with-counter with-menu">
            <a href="#">First level entry</a>

            <div class="app-navigation-entry-utils">
                <ul>
                    <li class="app-navigation-entry-utils-counter">15</li>
                    <li class="app-navigation-entry-utils-menu-button svg"><button></button></li>
                </ul>
            </div>

            <div class="app-navigation-entry-menu open">
                <ul>
```

```
                    <li><button class="icon-rename svg" title="rename"></button></li>
                    <li><button class="icon-delete svg" title="delete"></button></li>
                </ul>
            </div>

        </li>
    </ul>
</div>
```

The div with the class **app-navigation-entry-utils** contains only the button (class: **app-navigation-entry-utils-menu-button**) to display the menu but in many cases another entry is needed to display some sort of count (mails count, unread feed count, etc.). In that case add the **with-counter** class to the list entry to adjust the correct padding and text-oveflow of the entry's title.

The count should be limitted to 999 and turn to 999+ if any higher number is given. If AngularJS is used the following filter can be used to get the correct behaviour:

```
app.filter('counterFormatter', function () {
    'use strict';
    return function (count) {
        if (count > 999) {
            return '999+';
        }
        return count;
    };
});
```

Use it like this:

```
<li class="app-navigation-entry-utils-counter">{{ count | counterFormatter }}</li>
```

The menu is hidden by default (**display: none**) and has to be triggered by adding the **open** class to the **app-navigation-entry-menu** div.

In case of AngularJS the following small directive can be added to handle all the display and click logic out of the box:

```
app.run(function ($document, $rootScope) {
    'use strict';
    $document.click(function (event) {
        $rootScope.$broadcast('documentClicked', event);
    });
});

app.directive('appNavigationEntryUtils', function () {
    'use strict';
    return {
        restrict: 'C',
        link: function (scope, elm) {
            var menu = elm.siblings('.app-navigation-entry-menu');
            var button = $(elm)
                .find('.app-navigation-entry-utils-menu-button button');

            button.click(function () {
                menu.toggleClass('open');
            });

            scope.$on('documentClicked', function (scope, event) {
                if (event.target !== button[0]) {
                    menu.removeClass('open');
```

```
                }
            });
        }
    };
});
```

Editing

New in version 8.

Often an edit option is needed for an entry. To add one for a given entry simply hide the title and add the following div inside the entry:

```
<div id="app-navigation">
    <ul class="with-icon">
        <li>
            <a href="#" class="hidden">First level entry</a>

            <div class="app-navigation-entry-edit">
                <form>
                    <input type="text" value="First level entry" autofocus-on-insert>
                    <input type="submit" value="" class="action icon-checkmark svg">
                </form>
            </div>

        </li>
    </ul>
</div>
```

If AngularJS is used you want to autofocus the input box. This can be achieved by placing the show condition inside an **ng-if** on the **app-navigation-entry-edit** div and adding the following directive:

```
app.directive('autofocusOnInsert', function () {
    'use strict';
    return function (scope, elm) {
        elm.focus();
    };
});
```

ng-if is required because it removes/inserts the element into the DOM dynamically instead of just adding a **display: none** to it like **ng-show** and **ng-hide**.

Undo entry

New in version 8.

If you want to undo a performed action on a navigation entry such as deletion, you should show the undo directly in place of the entry and make it disappear after location change or 7 seconds:

```
<div id="app-navigation">
    <ul class="with-icon">
        <li>
            <a href="#" class="hidden">First level entry</a>

            <div class="app-navigation-entry-deleted">
                <div class="app-navigation-entry-deleted-description">Deleted X</div>
                <button class="app-navigation-entry-deleted-button icon-history svg" title="Undo"></b
```

```
            </div>
        </li>
    </ul>
</div>
```

Settings Area

To create a settings area create a div with the id **app-settings** inside the **app-navgiation** div:

```
<div id="app">

    <div id="app-navigation">

        <!-- Your navigation here -->

        <div id="app-settings">
            <div id="app-settings-header">
                <button class="settings-button"
                        data-apps-slide-toggle="#app-settings-content"
                ></button>
            </div>
            <div id="app-settings-content">
                <!-- Your settings in here -->
            </div>
        </div>
    </div>
</div>
```

The data attribute **data-apps-slide-toggle** slides up a target area using a jQuery selector and hides the area if the user clicks outside of it.

Icons

To use icons which are shipped in core, special classes to apply the background image are supplied. All of these classes use **background-position: center** and **background-repeat: no-repeat**.

- **icon-breadcrumb:**
- **icon-loading:** 
- **icon-loading-dark:** 
- **icon-loading-small:** 
- **icon-add:** 
- **icon-caret:** 
- **icon-caret-dark:** 

- **icon-checkmark:** 
- **icon-checkmark-white:**
- **icon-clock:** 
- **icon-close:** 
- **icon-confirm:** 
- **icon-delete:** 
- **icon-download:** 
- **icon-history:**
- **icon-info:** 
- **icon-lock:** 
- **icon-logout:** 
- **icon-mail:** 
- **icon-more:**
- **icon-password:** 
- **icon-pause:** 
- **icon-pause-big:** 
- **icon-play:** 
- **icon-play-add:** 
- **icon-play-big:** 
- **icon-play-next:** 
- **icon-play-previous:** 
- **icon-public:** 
- **icon-rename:** 
- **icon-search:** 
- **icon-settings:** 
- **icon-share:** 
- **icon-shared:** 

- **icon-sound:** 
- **icon-sound-off:** 
- **icon-star:**
- **icon-starred:** 
- **icon-toggle:** 
- **icon-triangle-e:** 
- **icon-triangle-n:** 
- **icon-triangle-s:** 
- **icon-upload:** 
- **icon-upload-white:**
- **icon-user:** 
- **icon-view-close:**
- **icon-view-next:**
- **icon-view-pause:**
- **icon-view-play:**
- **icon-view-previous:**
- **icon-calendar-dark:** 
- **icon-contacts-dark:** 
- **icon-file:** 
- **icon-files:** 
- **icon-folder:** 

- **icon-filetype-text:** 
- **icon-filetype-folder:** 
- **icon-home:** 
- **icon-link:** 
- **icon-music:** 
- **icon-picture:**

Translation

Nextcloud's translation system is powered by Transifex. To start translating sign up and enter a group. If your community app should be added to Transifex contact one of the translation team in the forums to set it up for you.

PHP

Should it ever be needed to use localized strings on the server-side, simply inject the L10N service from the Server-Container into the needed constructor

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Service\AuthorService;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('AuthorService', function($c) {
            return new AuthorService(
                $c->query('L10N')
            );
        });
```

```
        $container->registerService('L10N', function($c) {
            return $c->query('ServerContainer')->getL10N($c->query('AppName'));
        });
    }
}
```

Strings can then be translated in the following way:

```
<?php
namespace OCA\MyApp\Service;

use \OCP\IL10N;


class AuthorService {

    private $trans;

    public function __construct(IL10N $trans){
        $this->trans = $trans;
    }

    public function getLanguageCode() {
        return $this->trans->getLanguageCode();
    }

    public sayHello() {
        return $this->trans->t('Hello');
    }

    public function getAuthorName($name) {
        return $this->trans->t('Getting author %s', array($name));
    }

    public function getAuthors($count, $city) {
        return $this->trans->n(
            '%n author is currently in the city %s',  // singular string
            '%n authors are currently in the city %s',  // plural string
            $count,
            array($city)
        );
    }
}
```

Templates

In every template the global variable **$l** can be used to translate the strings using its methods **t()** and **n()**:

```
<div><?php p($l->t('Showing %s files', $_['count'])); ?></div>

<button><?php p($l->t('Hide')); ?></button>
```

JavaScript

There is a global function **t()** available for translating strings. The first argument is your app name, the second argument is the string to translate.

```
t('myapp', 'Hello World!');
```

For advanced usage, refer to the source code **core/js/l10n.js**, **t()** is bind to **OC.L10N.translate()**.

Hints

In case some translation strings may be translated wrongly because they have multiple meanings, you can add hints which will be shown in the Transifex web-interface:

```
<ul id="translations">
    <li id="add-new">
        <?php
            // TRANSLATORS Will be shown inside a popup and asks the user to add a new file
            p($l->t('Add new file'));
        ?>
    </li>
</ul>
```

Creating your own translatable files

If Transifex is not the right choice or the app is not accepted for translation, generate the gettext strings by yourself by creating an `l10n/` directory in the app folder and executing:

```
cd /srv/http/nextcloud/apps/myapp/l10n
perl l10n.pl read myapp
```

The translation script requires **Locale::PO** and **gettext**, installable via:

```
apt-get install liblocale-po-perl gettext
```

The above script generates a template that can be used to translate all strings of an app. This template is located in the folder `template/` with the name `myapp.pot`. It can be used by your favored translation tool which then creates a `.po` file. The `.po` file needs to be placed in a folder named like the language code with the app name as filename - for example `l10n/es/myapp.po`. After this step the perl script needs to be invoked to transfer the po file into our own fileformat that is more easily readable by the server code:

```
perl l10n.pl write myapp
```

Now the following folder structure is available:

```
myapp/l10n
|-- es
|   |-- myapp.po
|-- es.js
|-- es.json
|-- es.php
|-- l10n.pl
|-- templates
    |-- myapp.pot
```

You then just need the `.php`, `.json` and `.js` files for a working localized app.

Theming support

The Nextcloud theming app offers some tools for app developers to ensure that apps also match the themed look.

CSS classes

- **.nc-theming-main-background** Background in theming color
- **.nc-theming-main-text** Text in theming color
- **.nc-theming-contrast** Text in white/black color to be shown in front of the theming color

JavaScript

When the theming app is enabled, it provides the **OCA.Theming** object. It can be used to handle themed instances differently.

```
if(OCA.Theming) {
  $('.myapp-element').animate({backgroundColor:OCA.Theming.color});
}
```

The following information is available:

- **OCA.Theming.color** Main color
- **OCA.Theming.inverted** Will be true on bright theming colors to get contrast with text
- **OCA.Theming.name** Instance name
- **OCA.Theming.slogan** Instance slogan
- **OCA.Theming.url** Instance web address

Icons

The theming app will automatically generate favicons and home screen icons for each app by using the icon *img/app.svg* inside of the app folder. Any custom favicon set by an app will only be visible when the theming app is disabled.

Database Schema

Nextcloud uses a database abstraction layer on top of either PDO, depending on the availability of PDO on the server.

The database schema is inside `appinfo/database.xml` in MDB2's XML scheme notation where the placeholders *dbprefix* (*PREFIX* in your SQL) and *dbname* can be used for the configured database table prefix and database name.

An example database XML file would look like this:

```
<?xml version="1.0" encoding="UTF-8" ?>
<database>
 <name>*dbname*</name>
 <create>true</create>
 <overwrite>false</overwrite>
 <charset>utf8</charset>
 <table>
  <name>*dbprefix*yourapp_items</name>
  <declaration>
    <field>
      <name>id</name>
      <type>integer</type>
```

```
      <default>0</default>
      <notnull>true</notnull>
          <autoincrement>1</autoincrement>
      <length>4</length>
    </field>
    <field>
      <name>user</name>
      <type>text</type>
      <notnull>true</notnull>
      <length>64</length>
    </field>
    <field>
      <name>name</name>
      <type>text</type>
      <notnull>true</notnull>
      <length>100</length>
    </field>
    <field>
      <name>path</name>
      <type>clob</type>
      <notnull>true</notnull>
    </field>
  </declaration>
</table>
</database>
```

To update the tables used by the app, simply adjust the database.xml file and increase the app version number in `appinfo/info.xml` to trigger an update.

Database Access

The basic way to run a database query is to use the database connection provided by **OCP\IDBConnection**.

Inside your database layer class you can now start running queries like:

```
<?php
// db/authordao.php

namespace OCA\MyApp\Db;

use OCP\IDBConnection;

class AuthorDAO {

    private $db;

    public function __construct(IDBConnection $db) {
        $this->db = $db;
    }

    public function find($id) {
        $sql = 'SELECT * FROM `*PREFIX*myapp_authors` ' .
            'WHERE `id` = ?';
        $stmt = $this->db->prepare($sql);
        $stmt->bindParam(1, $id, \PDO::PARAM_INT);
        $stmt->execute();
```

```
        $row = $stmt->fetch();

        $stmt->closeCursor();
        return $row;
    }

}
```

Mappers

The aforementioned example is the most basic way to write a simple database query but the more queries amass, the more code has to be written and the harder it will become to maintain it.

To generalize and simplify the problem, split code into resources and create an **Entity** and a **Mapper** class for it. The mapper class provides a way to run SQL queries and maps the result onto the related entities.

To create a mapper, inherit from the mapper baseclass and call the parent constructor with the following parameters:

- Database connection
- Table name
- **Optional**: Entity class name, defaults to \OCA\MyApp\Db\Author in the example below

```
<?php
// db/authormapper.php

namespace OCA\MyApp\Db;

use OCP\IDBConnection;
use OCP\AppFramework\Db\Mapper;

class AuthorMapper extends Mapper {

    public function __construct(IDBConnection $db) {
        parent::__construct($db, 'myapp_authors');
    }


    /**
     * @throws \OCP\AppFramework\Db\DoesNotExistException if not found
     * @throws \OCP\AppFramework\Db\MultipleObjectsReturnedException if more than one result
     */
    public function find($id) {
        $sql = 'SELECT * FROM `*PREFIX*myapp_authors` ' .
            'WHERE `id` = ?';
        return $this->findEntity($sql, [$id]);
    }


    public function findAll($limit=null, $offset=null) {
        $sql = 'SELECT * FROM `*PREFIX*myapp_authors`';
        return $this->findEntities($sql, $limit, $offset);
    }


    public function authorNameCount($name) {
        $sql = 'SELECT COUNT(*) AS `count` FROM `*PREFIX*myapp_authors` ' .
```

```
            'WHERE `name` = ?';
        $stmt = $this->execute($sql, [$name]);

        $row = $stmt->fetch();
        $stmt->closeCursor();
        return $row['count'];
    }

}
```

Note: The cursor is closed automatically for all **INSERT**, **DELETE**, **UPDATE** queries and when calling the methods **findOneQuery**, **findEntities**, **findEntity**, **delete**, **insert** and **update**. For custom calls using execute you should always close the cursor after you are done with the fetching to prevent database lock problems on SqLite

Every mapper also implements default methods for deleting and updating an entity based on its id:

```
$authorMapper->delete($entity);
```

or:

```
$authorMapper->update($entity);
```

Entities

Entities are data objects that carry all the table's information for one row. Every Entity has an **id** field by default that is set to the integer type. Table rows are mapped from lower case and underscore separated names to pascal case attributes:

- **Table column name**: phone_number
- **Property name**: phoneNumber

```
<?php
// db/author.php
namespace OCA\MyApp\Db;

use OCP\AppFramework\Db\Entity;

class Author extends Entity {

    protected $stars;
    protected $name;
    protected $phoneNumber;

    public function __construct() {
        // add types in constructor
        $this->addType('stars', 'integer');
    }
}
```

Types

The following properties should be annotated by types, to not only assure that the types are converted correctly for storing them in the database (e.g. PHP casts false to the empty string which fails on postgres) but also for casting them when they are retrieved from the database.

The following types can be added for a field:

- integer
- float
- boolean

Accessing attributes

Since all attributes should be protected, getters and setters are automatically generated for you:

```
<?php
// db/author.php
namespace OCA\MyApp\Db;

use OCP\AppFramework\Db\Entity;

class Author extends Entity {
    protected $stars;
    protected $name;
    protected $phoneNumber;
}

$author = new Author();
$author->setId(3);
$author->getPhoneNumber()  // null
```

Custom attribute to database column mapping

By default each attribute will be mapped to a database column by a certain convention, e.g. **phoneNumber** will be mapped to the column **phone_number** and vice versa. Sometimes it is needed though to map attributes to different columns because of backwards compatibility. To define a custom mapping, simply override the **columnToProperty** and **propertyToColumn** methods of the entity in question:

```
<?php
// db/author.php
namespace OCA\MyApp\Db;

use OCP\AppFramework\Db\Entity;

class Author extends Entity {
    protected $stars;
    protected $name;
    protected $phoneNumber;

    // map attribute phoneNumber to the database column phonenumber
    public function columnToProperty($column) {
        if ($column === 'phonenumber') {
            return 'phoneNumber';
        } else {
            return parent::columnToProperty($column);
        }
    }

    public function propertyToColumn($property) {
        if ($column === 'phoneNumber') {
```

```
            return 'phonenumber';
        } else {
            return parent::propertyToColumn($property);
        }
    }

}
```

Slugs

Slugs are used to identify resources in the URL by a string rather than integer id. Since the URL allows only certain values, the entity baseclass provides a slugify method for it:

```
<?php
$author = new Author();
$author->setName('Some*thing');
$author->slugify('name');  // Some-thing
```

Configuration

The config that allows the app to set global, app and user settings can be injected from the ServerContainer. All values are saved as strings and must be cast to the correct value.

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Service\AuthorService;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('AuthorService', function($c) {
            return new AuthorService(
                $c->query('Config'),
                $c->query('AppName')
            );
        });

        $container->registerService('Config', function($c) {
            return $c->query('ServerContainer')->getConfig();
        });
    }
}
```

System values

System values are saved in the `config/config.php` and allow the app to modify and read the global configuration:

```
<?php
namespace OCA\MyApp\Service;

use \OCP\IConfig;


class AuthorService {

    private $config;
    private $appName;

    public function __construct(IConfig $config, $appName){
        $this->config = $config;
        $this->appName = $appName;
    }

    public function getSystemValue($key) {
        return $this->config->getSystemValue($key);
    }

    public function setSystemValue($key, $value) {
        $this->config->setSystemValue($key, $value);
    }

}
```

App values

App values are saved in the database per app and are useful for setting global app settings:

```
<?php
namespace OCA\MyApp\Service;

use \OCP\IConfig;


class AuthorService {

    private $config;
    private $appName;

    public function __construct(IConfig $config, $appName){
        $this->config = $config;
        $this->appName = $appName;
    }

    public function getAppValue($key) {
        return $this->config->getAppValue($this->appName, $key);
    }

    public function setAppValue($key, $value) {
        $this->config->setAppValue($this->appName, $key, $value);
    }
```

```
}
```

User values

User values are saved in the database per user and app and are good for saving user specific app settings:

```
<?php
namespace OCA\MyApp\Service;

use \OCP\IConfig;


class AuthorService {

    private $config;
    private $appName;

    public function __construct(IConfig $config, $appName){
        $this->config = $config;
        $this->appName = $appName;
    }

    public function getUserValue($key, $userId) {
        return $this->config->getUserValue($userId, $this->appName, $key);
    }

    public function setUserValue($key, $userId, $value) {
        $this->config->setUserValue($userId, $this->appName, $key, $value);
    }

}
```

Filesystem

Because users can choose their storage backend, the filesystem should be accessed by using the appropriate filesystem classes.

Filesystem classes can be injected from the ServerContainer by calling the method **getRootFolder()**, **getUserFolder()** or **getAppFolder()**:

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Storage\AuthorStorage;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();
```

```
        /**
         * Storage Layer
         */
        $container->registerService('AuthorStorage', function($c) {
            return new AuthorStorage($c->query('RootStorage'));
        });

        $container->registerService('RootStorage', function($c) {
            return $c->query('ServerContainer')->getRootFolder();
        });

    }
}
```

Writing to a file

All methods return a Folder object on which files and folders can be accessed, or filesystem operations can be performed relatively to their root. For instance for writing to file:*nextcloud/data/myfile.txt* you should get the root folder and use:

```
<?php
namespace OCA\MyApp\Storage;

class AuthorStorage {

    private $storage;

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

    public function writeTxt($content) {
        // check if file exists and write to it if possible
        try {
            try {
                $file = $this->storage->get('/myfile.txt');
            } catch(\OCP\Files\NotFoundException $e) {
                $this->storage->touch('/myfile.txt');
                $file = $this->storage->get('/myfile.txt');
            }

            // the id can be accessed by $file->getId();
            $file->putContent($content);

        } catch(\OCP\Files\NotPermittedException $e) {
            // you have to create this exception by yourself ;)
            throw new StorageException('Cant write to file');
        }
    }
}
```

Reading from a file

Files and folders can also be accessed by id, by calling the **getById** method on the folder.

```php
<?php
namespace OCA\MyApp\Storage;

class AuthorStorage {

    private $storage;

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

    public function getContent($id) {
        // check if file exists and read from it if possible
        try {
            $file = $this->storage->getById($id);
            if($file instanceof \OCP\Files\File) {
                return $file->getContent();
            } else {
                throw new StorageException('Can not read from folder');
            }
        } catch(\OCP\Files\NotFoundException $e) {
            throw new StorageException('File does not exist');
        }
    }
}
```

AppData

Often an app wants to store data. However not all data that is stored belongs with the users files. Often you just want a very simple storage to have some temp files. In order to facilitate this there is the AppData folder that provides each app with a private simple filesystem.

Usage is almost trivial when your app is using the AppFramework.

```php
<?php

namespace OCA\MyApp\Controller\MyController;

use OCP\AppFramework\Controller;
use OCP\Files\IAppData;
use OCP\IRequest;

class MyController extends Controller {
    /** @var IAppData */
    private $appData;

    public function __construct($appName,
                                IRequest $request,
                                IAppData $appData) {
        parent::__construct($appName, $request);
        $this->appData = $appData;
    }
}
```

This gives your controller access to the IAppData simple filesystem of your app.

The Simple Filesystem

The *IAppData* uses the simple filesystem. This is a very simplified filesystem that will allow for easy mapping to for example memcaches. The filesystem has three elements: *root*, *folder*, *file*.

The *root* can only contain folders. And each folder can only contain files. This is limited to keep things simple and to allow easy mapping to other backends. For example a sysadmin might chose to map the avatars to fast storage since they are used often.

Usermanagement

Users can be managed using the UserManager which is injected from the ServerContainer:

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Service\UserService;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('UserService', function($c) {
            return new UserService(
                $c->query('UserManager')
            );
        });

        $container->registerService('UserManager', function($c) {
            return $c->query('ServerContainer')->getUserManager();
        });
    }
}
```

Creating users

Creating a user is done by passing a username and password to the create method:

```
<?php
namespace OCA\MyApp\Service;

class UserService {

    private $userManager;

    public function __construct($userManager){
```

```
        $this->userManager = $userManager;
    }

    public function create($userId, $password) {
        return $this->userManager->create($userId, $password);
    }

}
```

Modifying users

Users can be modified by getting a user by the userId or by a search pattern. The returned user objects can then be used to:

- Delete them
- Set a new password
- Disable/Enable them
- Get their home directory

```
<?php
namespace OCA\MyApp\Service;

class UserService {

    private $userManager;

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

    public function delete($userId) {
        return $this->userManager->get($userId)->delete();
    }

    // recoveryPassword is used for the encryption app to recover the keys
    public function setPassword($userId, $password, $recoveryPassword) {
        return $this->userManager->get($userId)->setPassword($password, $recoveryPassword);
    }

    public function disable($userId) {
        return $this->userManager->get($userId)->setEnabled(false);
    }

    public function getHome($userId) {
        return $this->userManager->get($userId)->getHome();
    }
}
```

User Session Information

To login, logout or getting the currently logged in user, the UserSession has to be injected from the ServerContainer:

```
<?php
namespace OCA\MyApp\AppInfo;
```

```
use \OCP\AppFramework\App;

use \OCA\MyApp\Service\UserService;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('UserService', function($c) {
            return new UserService(
                $c->query('UserSession')
            );
        });

        $container->registerService('UserSession', function($c) {
            return $c->query('ServerContainer')->getUserSession();
        });

        // currently logged in user, userId can be gotten by calling the
        // getUID() method on it
        $container->registerService('User', function($c) {
            return $c->query('UserSession')->getUser();
        });
    }
}
```

Then users can be logged in by using:

```
<?php
namespace OCA\MyApp\Service;

class UserService {

    private $userSession;

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

    public function login($userId, $password) {
        return $this->userSession->login($userId, $password);
    }

    public function logout() {
        $this->userSession->logout();
    }

}
```

Two-factor Providers

Two-factor auth providers apps are used to plug custom second factors into the Nextcloud core. The following code was taken from the two-factor test app.

Implementing a simple two-factor auth provider

Two-factor auth providers must implement the `OCP\Authentication\TwoFactorAuth\IProvider` interface. The example below shows a minimalistic example of such a provider.

```php
<?php

namespace OCA\TwoFactor_Test\Provider;

use OCP\Authentication\TwoFactorAuth\IProvider;
use OCP\IUser;
use OCP\Template;

class TwoFactorTestProvider implements IProvider {

	/**
	 * Get unique identifier of this 2FA provider
	 *
	 * @return string
	 */
	public function getId() {
		return 'test';
	}

	/**
	 * Get the display name for selecting the 2FA provider
	 *
	 * @return string
	 */
	public function getDisplayName() {
		return 'Test';
	}

	/**
	 * Get the description for selecting the 2FA provider
	 *
	 * @return string
	 */
	public function getDescription() {
		return 'Use a test provider';
	}

	/**
	 * Get the template for rending the 2FA provider view
	 *
	 * @param IUser $user
	 * @return Template
	 */
	public function getTemplate(IUser $user) {
		// If necessary, this is also the place where you might want
		// to send out a code via e-mail or SMS.
```

```
                // 'challenge' is the name of the template
                return new Template('twofactor_test', 'challenge');
        }

        /**
         * Verify the given challenge
         *
         * @param IUser $user
         * @param string $challenge
         */
        public function verifyChallenge(IUser $user, $challenge) {
                if ($challenge === 'passme') {
                        return true;
                }
                return false;
        }

        /**
         * Decides whether 2FA is enabled for the given user
         *
         * @param IUser $user
         * @return boolean
         */
        public function isTwoFactorAuthEnabledForUser(IUser $user) {
                // 2FA is enforced for all users
                return true;
        }

}
```

Registering a two-factor auth provider

You need to inform the Nextcloud core that the app provides two-factor auth functionality. Two-factor providers are registered via `info.xml`.

```
<two-factor-providers>
        <provider>OCA\TwoFactor_Test\Provider\TwoFactorTestProvider</provider>
</two-factor-providers>
```

Hooks

Hooks are used to execute code before or after an event has occurred. This is for instance useful to run cleanup code after users, groups or files have been deleted. Hooks should be registered in the app.php:

```
<?php
namespace OCA\MyApp\AppInfo;

$app = new Application();
$app->getContainer()->query('UserHooks')->register();
```

The hook logic should be in a separate class that is being registered in the Container

```
<?php
namespace OCA\MyApp\AppInfo;
```

```
use \OCP\AppFramework\App;

use \OCA\MyApp\Hooks\UserHooks;


class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('UserHooks', function($c) {
            return new UserHooks(
                $c->query('ServerContainer')->getUserManager()
            );
        });
    }
}
```

```
<?php
namespace OCA\MyApp\Hooks;
use OCP\IUserManager;

class UserHooks {

    private $userManager;

    public function __construct(IUserManager $userManager){
        $this->userManager = $userManager;
    }

    public function register() {
        $callback = function($user) {
            // your code that executes before $user is deleted
        };
        $this->userManager->listen('\OC\User', 'preDelete', $callback);
    }

}
```

Available hooks

The scope is the first parameter that is passed to the **listen** method, the second parameter is the method and the third one the callback that should be executed once the hook is being called, e.g.:

```
<?php

// listen on user predelete
$callback = function($user) {
    // your code that executes before $user is deleted
};
$userManager->listen('\OC\User', 'preDelete', $callback);
```

Hooks can also be removed by using the **removeListener** method on the object:

```
<?php

// delete previous callback
$userManager->removeListener(null, null, $callback);
```

The following hooks are available:

Session

Injectable from the ServerContainer by calling the method **getUserSession()**.

Hooks available in scope **\OC\User**:

- **preSetPassword** (\OC\User\User $user, string $password, string $recoverPassword)
- **postSetPassword** (\OC\User\User $user, string $password, string $recoverPassword)
- **preDelete** (\OC\User\User $user)
- **postDelete** (\OC\User\User $user)
- **preCreateUser** (string $uid, string $password)
- **postCreateUser** (\OC\User\User $user)
- **preLogin** (string $user, string $password)
- **postLogin** (\OC\User\User $user)
- **logout** ()

UserManager

Injectable from the ServerContainer by calling the method **getUserManager()**.

Hooks available in scope **\OC\User**:

- **preSetPassword** (\OC\User\User $user, string $password, string $recoverPassword)
- **postSetPassword** (\OC\User\User $user, string $password, string $recoverPassword)
- **preDelete** (\OC\User\User $user)
- **postDelete** (\OC\User\User $user)
- **preCreateUser** (string $uid, string $password)
- **postCreateUser** (\OC\User\User $user, string $password)

GroupManager

Hooks available in scope **\OC\Group**:

- **preAddUser** (\OC\Group\Group $group, \OC\User\User $user)
- **postAddUser** (\OC\Group\Group $group, \OC\User\User $user)
- **preRemoveUser** (\OC\Group\Group $group, \OC\User\User $user)
- **postRemoveUser** (\OC\Group\Group $group, \OC\User\User $user)
- **preDelete** (\OC\Group\Group $group)

- **postDelete** (\OC\Group\Group $group)
- **preCreate** (string $groupId)
- **postCreate** (\OC\Group\Group $group)

Filesystem Root

Injectable from the ServerContainer by calling the method **getRootFolder()**, **getUserFolder()** or **getAppFolder()**.

Filesystem hooks available in scope **\OC\Files**:

- **preWrite** (\OCP\Files\Node $node)
- **postWrite** (\OCP\Files\Node $node)
- **preCreate** (\OCP\Files\Node $node)
- **postCreate** (\OCP\Files\Node $node)
- **preDelete** (\OCP\Files\Node $node)
- **postDelete** (\OCP\Files\Node $node)
- **preTouch** (\OCP\Files\Node $node, int $mtime)
- **postTouch** (\OCP\Files\Node $node)
- **preCopy** (\OCP\Files\Node $source, \OCP\Files\Node $target)
- **postCopy** (\OCP\Files\Node $source, \OCP\Files\Node $target)
- **preRename** (\OCP\Files\Node $source, \OCP\Files\Node $target)
- **postRename** (\OCP\Files\Node $source, \OCP\Files\Node $target)

Filesystem Scanner

Filesystem scanner hooks available in scope **\OC\Files\Utils\Scanner**:

- **scanFile** (string $absolutePath)
- **scanFolder** (string $absolutePath)
- **postScanFile** (string $absolutePath)
- **postScanFolder** (string $absolutePath)

Background Jobs (Cron)

Background/cron jobs are usually registered in the `appinfo/app.php` by using the **addRegularTask** method, the class and the method to run:

```
<?php
\OCP\Backgroundjob::addRegularTask('\OCA\MyApp\Cron\SomeTask', 'run');
```

The class for the above example would live in `cron/sometask.php`. Try to keep the method as small as possible because its hard to test static methods. Simply reuse the app container and execute a service that was registered in it.

```php
<?php
namespace OCA\MyApp\Cron;

use \OCA\MyApp\AppInfo\Application;

class SomeTask {

    public static function run() {
        $app = new Application();
        $container = $app->getContainer();
        $container->query('SomeService')->run();
    }

}
```

Don't forget to configure the cron service on the server by executing:

```
sudo crontab -u http -e
```

where **http** is your Web server user, and add:

```
*/15  *  *  *  * php -f /srv/http/nextcloud/cron.php
```

Settings

An app can register both admin settings as well as personal settings.

Admin

For Nextcloud 10 the admin settings page got reworked. It is not a long list anymore, but divided into sections, where related settings forms are grouped. For example, in the **Sharing** section are only settings (built-in and of apps) relate to sharing.

Settings Form

For the settings form, three things are necessary

1. A class implementing `\OCP\Settings\ISettings`
2. A template
3. The implementing class specified in the apps's info.xml

Below is an example for an implementor of the ISettings interface. It is based on the survey_client solution.

```php
<?php
namespace OCA\YourAppNamespace\Settings;

use OCA\YourAppNamespace\Collector;
use OCP\AppFramework\Http\TemplateResponse;
use OCP\BackgroundJob\IJobList;
use OCP\IConfig;
use OCP\IDateTimeFormatter;
use OCP\IL10N;
use OCP\Settings\ISettings;
```

```php
class AdminSettings implements ISettings {

	/** @var Collector */
	private $collector;

	/** @var IConfig */
	private $config;

	/** @var IL10N */
	private $l;

	/** @var IDateTimeFormatter */
	private $dateTimeFormatter;

	/** @var IJobList */
	private $jobList;

	/**
	 * Admin constructor.
	 *
	 * @param Collector $collector
	 * @param IConfig $config
	 * @param IL10N $l
	 * @param IDateTimeFormatter $dateTimeFormatter
	 * @param IJobList $jobList
	 */
	public function __construct(Collector $collector,
	                            IConfig $config,
	                            IL10N $l,
	                            IDateTimeFormatter $dateTimeFormatte
	                            IJobList $jobList
	) {
		$this->collector = $collector;
		$this->config = $config;
		$this->l = $l;
		$this->dateTimeFormatter = $dateTimeFormatter;
		$this->jobList = $jobList;
	}

	/**
	 * @return TemplateResponse
	 */
	public function getForm() {

		$lastSentReportTime = (int) $this->config->getAppValue('survey_client', 'last_sent',
		if ($lastSentReportTime === 0) {
			$lastSentReportDate = $this->l->t('Never');
		} else {
			$lastSentReportDate = $this->dateTimeFormatter->formatDate($lastSentReportTi
		}

		$lastReport = $this->config->getAppValue('survey_client', 'last_report', '');
		if ($lastReport !== '') {
			$lastReport = json_encode(json_decode($lastReport, true), JSON_PRETTY_PRINT);
		}

		$parameters = [
			'is_enabled' => $this->jobList->has('OCA\Survey_Client\BackgroundJobs\Monthly
```

```
                        'last_sent' => $lastSentReportDate,
                        'last_report' => $lastReport,
                        'categories' => $this->collector->getCategories()
                ];

                return new TemplateResponse('yourappid', 'admin', $parameters);
        }

        /**
         * @return string the section ID, e.g. 'sharing'
         */
        public function getSection() {
                return 'survey_client';
        }

        /**
         * @return int whether the form should be rather on the top or bottom of
         * the admin section. The forms are arranged in ascending order of the
         * priority values. It is required to return a value between 0 and 100.
         */
        public function getPriority() {
                return 50;
        }

}
```

The parameters of the constructor will be resolved and an instance created automatically on demand, so that the developer does not need to take care of it.

`getSection` is supposed to return the section ID of the desired admin section. Currently, built-in values are `server`, `sharing`, `encryption`, `logging`, `additional` and `tips-tricks`. Apps can register sections of their own (see below), and also register into sections of other apps.

`getPriority` is used to order forms within a section. The lower the value, the more on top it will appear, and vice versa. The result depends on the priorities of other settings.

Nextcloud will look for the templates in a template folder located in your apps root directory. It should always end on .php, in this case `templates/admin.php` would be the final relative path.

```
<?php
/** @var $l \OCP\IL10N */
/** @var $_ array */

script('myappid', 'admin');         // adds a Javascript file
style('survey_client', 'admin');    // adds a CSS file
?>

<div id="survey_client" class="section">
        <h2><?php p($l->t('Your app')); ?></h2>

        <p>
                <?php p($l->t('Only administrators are allowed to click the red button')); ?>
        </p>

        <button><?php p($l->t('Click red button')); ?></button>

        <p>
                <input id="your_app_magic" name="your_app_magic"
                        type="checkbox" class="checkbox" value="1" <?php if ($_['is_enabled']): ?
```

```
                <label for="your_app_magic"><?php p($l->t('Do some magic')); ?></label>
        </p>

        <h3><?php p($l->t('Things to define')); ?></h3>
        <?php
        foreach ($_['categories'] as $category => $data) {
                ?>
                <p>
                        <input id="your_app_<?php p($category); ?>" name="your_app_<?php p($category)
                                type="checkbox" class="checkbox your_app_category" value="1" <?php
                        <label for="your_app_<?php p($category); ?>"><?php print_unescaped($data['dis
                </p>
                <?php
        }
        ?>

        <?php if (!empty($_['last_report'])): ?>

        <h3><?php p($l->t('Last report')); ?></h3>

        <p><textarea title="<?php p($l->t('Last report')); ?>" class="last_report" readonly="readonly

        <em class="last_sent"><?php p($l->t('Sent on: %s', [$_['last_sent']])); ?></em>

        <?php endif; ?>

</div>
```

Then, the implementing class should be added to the info.xml. Settings will be registered upon install and update. When settings are added to an existing, installed and enabled app, it should be made sure that the version is increased so Nextcloud can register the class. It is only possible to register one ISettings implementor.

For a more complex example using embedded templates have a look at the implementation of the **user_ldap** app.

Section

It is also possible, that an app registers its own section. This should be done only, if there is not fitting corresponding section and the apps settings form takes a lot of screen estate. Otherwise, register to "additional".

Basically, it works the same way as with the settings form. There are only two differences. First, the interface that must be implemented is `\OCP\Settings\ISection`.

Second, a template is not necessary.

An example implementation of the ISection interface:

```
<?php
namespace OCA\YourAppNamespace\Settings;

use OCP\IL10N;
use OCP\Settings\ISection;

class AdminSection implements ISection {

        /** @var IL10N */
        private $l;

        public function __construct(IL10N $l) {
```

```
                $this->l = $l;
        }

        /**
         * returns the ID of the section. It is supposed to be a lower case string
         *
         * @returns string
         */
        public function getID() {
                return 'yourappid'; //or a generic id if feasible
        }

        /**
         * returns the translated name as it should be displayed, e.g. 'LDAP / AD
         * integration'. Use the L10N service to translate it.
         *
         * @return string
         */
        public function getName() {
                return $this->l->t('Translatable Section Name');
        }

        /**
         * @return int whether the form should be rather on the top or bottom of
         * the settings navigation. The sections are arranged in ascending order of
         * the priority values. It is required to return a value between 0 and 99.
         */
        public function getPriority() {
                return 80;
        }

}
```

Also the section must be registered in the apps's info.xml.

Personal

Registering personal settings follows and old style yet. Within the app intialisation (e.g. in appinfo/app.php) a method must be called:

```
<?php
\OCP\App::registerPersonal('yourappid', 'personal');
```

Upon opening the personal page, Nextcloud will look for `personal.php` script, execute it and print the output.

Logging

The logger can be injected from the ServerContainer:

```
<?php
namespace OCA\MyApp\AppInfo;

use \OCP\AppFramework\App;

use \OCA\MyApp\Service\AuthorService;
```

```
class Application extends App {

    public function __construct(array $urlParams=array()){
        parent::__construct('myapp', $urlParams);

        $container = $this->getContainer();

        /**
         * Controllers
         */
        $container->registerService('AuthorService', function($c) {
            return new AuthorService(
                $c->query('Logger'),
                $c->query('AppName')
            );
        });

        $container->registerService('Logger', function($c) {
            return $c->query('ServerContainer')->getLogger();
        });
    }
}
```

and then be used in the following way:

```
<?php
namespace OCA\MyApp\Service;

use \OCP\ILogger;


class AuthorService {

    private $logger;
    private $appName;

    public function __construct(ILogger $logger, $appName){
        $this->logger = $logger;
        $this->appName = $appName;
    }

    public function log($message) {
        $this->logger->error($message, array('app' => $this->appName));
    }

}
```

The following methods are available:

- **emergency**
- **alert**
- **critical**
- **error**
- **warning**
- **notice**

- **info**
- **debug**

Testing

All PHP classes can be tested with PHPUnit, JavaScript can be tested by using Karma.

PHP

The PHP tests go into the **tests/** directory and PHPUnit can be run with:

```
phpunit tests/
```

When writing your own tests, please ensure that PHPUnit bootstraps from `tests/bootstrap.php`, to set up various environment variables and autoloader registration correctly. Without this, you will see errors as the Nextcloud autoloader security policy prevents access to the tests/ subdirectory. This can be configured in your `phpunit.xml` file as follows:

```
<phpunit bootstrap="../../tests/bootstrap.php">
```

PHP classes should be tested by accessing them from the container to ensure that the container is wired up properly. Services that should be mocked can be replaced directly in the container.

A test for the **AuthorStorage** class in Filesystem:

```
<?php
namespace OCA\MyApp\Storage;

class AuthorStorage {

    private $storage;

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

    public function getContent($id) {
        // check if file exists and write to it if possible
        try {
            $file = $this->storage->getById($id);
            if($file instanceof \OCP\Files\File) {
                return $file->getContent();
            } else {
                throw new StorageException('Can not read from folder');
            }
        } catch(\OCP\Files\NotFoundException $e) {
            throw new StorageException('File does not exist');
        }
    }
}
```

would look like this:

```
<?php
// tests/Storage/AuthorStorageTest.php
namespace OCA\MyApp\Tests\Storage;
```

```
class AuthorStorageTest extends \Test\TestCase {

    private $container;
    private $storage;

    protected function setUp() {
        parent::setUp();

        $app = new \OCA\MyApp\AppInfo\Application();
        $this->container = $app->getContainer();
        $this->storage = $storage = $this->getMockBuilder('\OCP\Files\Folder')
            ->disableOriginalConstructor()
            ->getMock();

        $this->container->registerService('RootStorage', function($c) use ($storage) {
            return $storage;
        });
    }

    /**
     * @expectedException \OCA\MyApp\Storage\StorageException
     */
    public function testFileNotFound() {
        $this->storage->expects($this->once())
            ->method('get')
            ->with($this->equalTo(3))
            ->will($this->throwException(new \OCP\Files\NotFoundException()));

        $this->container['AuthorStorage']->getContent(3);
    }

}
```

Make sure to extend the `\Test\TestCase` class with your test and always call the parent methods, when overwriting `setUp()`, `setUpBeforeClass()`, `tearDown()` or `tearDownAfterClass()` method from the TestCase. These methods set up important stuff and clean up the system after the test, so the next test can run without side effects, like remaining files and entries in the file cache, etc.

App store publishing

The Nextcloud App Store

The Nextcloud app store is build into Nextcloud to allow you to get your apps to users as easily and safely as possible. The app store and the process of publishing apps aims to be:

- secure
- transparent
- welcoming
- fair
- easy to maintain

Apps in the store are divided in three 'levels' of trust:

- Official
- Approved
- Experimental

With each level come requirements and a position in the store.

Official

Official apps are developed by and within the Nextcloud community and its Github repository and offer functionality central to Nextcloud. They are ready for serious use and can be considered a part of Nextcloud.

Requirements:

- developed in Nextcloud github repo
- minimum of 2 active maintainers and contributions from others
- security audited and design reviewed
- app is at least 6 months old and has seen regular releases
- follows app guidelines
- supports the same platforms and technologies mentioned in the release notes of the Nextcloud version this app is made for

App store:

- available in Apps page in separate category
- sorted first in all overviews, 'Official' tag
- shown as featured, on nextcloud.com etc
- major releases optionally featured on nextcloud.com
- new versions/updates approved by at least one other person

note: Official apps include those that are part of the release tarball. We'd like to keep the tarball minimal so most official apps are not part of the standard installation.

Approved

Approved apps are developed by trusted developers and have passed a cursory security check. They are actively maintained in an open code repository and their maintainers deem them to be stable for casual to normal use.

Requirements:

- code is developed in an open and version-managed code repository, ideally github with git but other scm/hosting is OK.
- minimum of one active developer/maintainer
- minimum 5 ratings, average score 60/100 or better
- app is at least 3 months old
- follows app guidelines
- the developer is trusted
- app is subject to unannounced security audits

- has defined requirements and dependencies (like what browsers, databases, PHP versions and so on are supported)

Note: **Developer trust**: The developer(s) is/are known in community; he/she has/have been active for a while, have met others at events and/or worked with others in various areas.

Note: **security audits**: in practice this means that at least some of the code of this developer has been audited; either through another app by the same developer or with an earlier version of the app. And that the attitude of the developer towards these audits has been positive.

App store:

- visible in app store by default
- sorted above experimental apps
- search results sorted by ratings
- developer can directly push new versions to the store
- warning shows for security/stability risks

Experimental

Apps which have not been checked at all for security and/or are new, known to be unstable or under heavy development.

Requirements:

- no malicious intent found from this developer at any time
- 0 confirmed security problems
- less than 3 unconfirmed 'security flags'
- rating over 20/100

App store:

- show up in Apps page provided user has enabled "allow installation of experimental apps" in the settings.
- Warning about security and stability risks is shown for app
- sorted below all others.

Getting an app approved

If you want your app to be approved, make sure you fulfill all the requirements and then create an issue in the app approval github repository using this template. A team of Nextcloud contributors will review your application. Updates to an app require re-review but, of course, an initial review takes more effort and time than the checking of an update.

You are encouraged to help review other contributors' apps as well! Every app requires at least two independent reviews so your review of at least 2 (more is better!) other apps will ensure the process continues smoothly. Thank you for participating in this process and being a great Nextcloud Community member!

Using the code checker

Before asking for approval, it is best to check your app code with the code checker, and fix the issues found by the code checker.

```
./occ app:check-code <app_name>
```

Losing a rating

Apps can lose their rating when:

- they are found to no longer satisfy the requirements
- when security/malicious intent issues are found
- when a developer requests so

App guidelines

These are the app guidelines an app has to comply with to have a chance to be approved.

Legal and security

- Apps can not use 'Nextcloud' in their name
- Irregular and unannounced security audits of all apps can and will take place.
- **If any indication of malicious intent or bad faith is found the developer(s) in question can count on a minimum 2 year ban**
  - Malicious intent includes deliberate spying on users by leaking user data to a third party system or adding a back door (like a hard-coded user account) to Nextcloud. An unintentional security bug that gets fixed in time won't be considered bad faith.
- Apps do not violate any laws; it has to comply with copyright- and trademark law.
- App authors have to respond timely to security concerns and not make Nextcloud more vulnerable to attack.

Note: distributing malicious or illegal applications can have legal consequences including, but not limited to Nextcloud or affected users taking legal action.

Be technically sound

- Apps can only use the public Nextcloud API
- At time of the release of an app it can only be configured to be compatible with the latest Nextcloud release +1
- Apps should not cause Nextcloud to break, consume excessive memory or slow Nextcloud down
- Apps should not hamper functionality of Nextcloud unless that is explicitly the goal of the app

Respect the users

- Apps have to follow design and HTML/CSS layout guidelines
- Apps correctly clean up after themselves on uninstall and correctly handle up- and downgrades
- Apps clearly communicate their intended purpose and active features, including features introduced through updates.
- Apps respect the users' choices and do not make unexpected changes, or limit users' ability to revert them. For example, they do not remove other apps or disable settings.
- Apps must respect user privacy. IF user data is sent anywhere, this must be clearly explained and be kept to a minimum for the functioning of an app. Use proper security measures when needed.
- App authors must provide means to contact them, be it through a bug tracker, forum or mail.

Apps which break the guidelines will lose their 'approved' or 'official' state; and might be blocked from the app store altogether. This also has repercussions for the author, especially in case of security concerns, he/she might find themselves blocked from submitting applications.

Code Signing

Nextcloud supports code signing for the core releases, and for Nextcloud applications. Code signing gives our users an additional layer of security by ensuring that nobody other than authorized persons can push updates.

It also ensures that all upgrades have been executed properly, so that no files are left behind, and all old files are properly replaced. In the past, invalid updates were a significant source of errors when updating Nextcloud.

FAQ

Why Did Nextcloud Add Code Signing?

By supporting Code Signing we add another layer of security by ensuring that nobody other than authorized persons can push updates for applications, and ensuring proper upgrades.

Do We Lock Down Nextcloud?

The Nextcloud project is open source and always will be. We do not want to make it more difficult for our users to run Nextcloud. Any code signing errors on upgrades will not prevent Nextcloud from running, but will display a warning on the Admin page. For applications that are not tagged "Official" the code signing process is optional.

Not Open Source Anymore?

The Nextcloud project is open source and always will be. The code signing process is optional, though highly recommended. The code check for the core parts of Nextcloud is enabled when the Nextcloud release version branch has been set to stable.

For custom distributions of Nextcloud it is recommended to change the release version branch in version.php to something else than "stable".

Is Code Signing Mandatory For Apps?

Code signing is optional for all third-party applications. Applications with a tag of "Official" on apps.owncloud.com require code signing.

Technical details

Nextcloud uses a X.509 based approach to handle authentication of code. Each Nextcloud release contains the certificate of a shipped Nextcloud Code Signing Root Authority. The private key of this certificate is only accessible to the project leader, who may grant trusted project members with a copy of this private key.

This Root Authority is only used for signing certificate signing requests (CSRs) for additional certificates. Certificates issued by the Root Authority must always to be limited to a specific scope, usually the application identifier. This enforcement is done using the `CN` attribute of the certificate.

Code signing is then done by creating a `signature.json` file with the following content:

```
{
    "hashes": {
        "/filename.php":
        "2401fed2eea6f2c1027c482a633e8e25cd46701f811e2d2c10dc213fd95fa60e350b
        ccbbebdccc73a042b1a2799f673fbabadc783284cc288e4f1a1eacb74e3d",
        "/lib/base.php":
        "55548cc16b457cd74241990cc9d3b72b6335f2e5f45eee95171da024087d114fcbc2
        effc3d5818a6d5d55f2ae960ab39fd0414d0c542b72a3b9e08eb21206dd9"
    },
    "certificate": "-----BEGIN CERTIFICATE-----
    MIIBvTCCASagAwIBAgIUPvawyqJwCwYazcv7iz16TWxfeUMwDQYJKoZIhvcNAQEF\
    nBQAwIzEhMB8GA1UECgwYb3duQ2xvdWQgQ29kZSBTaWduaW5nIENBMB4XDTE1MTAx\
    nNDEzMTcxMFoXDTE2MTAxNDEzMTcxMFowEzERMA8GA1UEAwwIY29udGFjdHMwgZ8w\
    nDQYJKoZIhvcNAQEBBQADgY0AMIGJAoGBANoQesGdCW0L2L+a2xITYipixkScrIpB\
    nkX5Snu3fs45MscDb61xByjBSlFgR4QI6McoCipPw4SUr28EaExVvgPSvqUjYLGps\
    nfiv0Cvgquzbx/X3mUcdk9LcFo1uWGtrTfkuXSKX41PnJGTr6RQWGIBd1V52q1qbC\
    nJKkfzyeMeuQfAgMBAAEwDQYJKoZIhvcNAQEFBQADgYEAvF/KIhRMQ3tYTmgHWsiM\
    nwDMgIDb7iaHF0fS+/Nvo4PzoTO/trev6tMyjLbJ7hgdCpz/1sNzE11Cibf6V6dsz\
    njCE9invP368Xv0bTRObRqeSNsGogGl5ceAvR0c9BG+NRIKHcly3At3gLkS2791bC\
    niG+UxI/MNcWV0uJg9S63LF8=\n
    -----END CERTIFICATE-----",
    "signature": "U29tZVNpZ25lZERhdGFFeGFtcGxl"
}
```

`hashes` is an array of all files in the folder with their corresponding SHA-512 hashes. `certificate` is the certificate used for signing. It has to be issued by the Nextcloud Root Authority, and its CN needs to be permitted to perform the required action. The `signature` is then a signature of the hashes which can be verified using the certificate.

Having the certificate bundled within the `signature.json` file has the advantage that even if a developer loses their certificate, future updates can still be ensured by having a new certificate issued.

How Code Signing Affects Apps in the App Store

- Apps which have an `official` tag **MUST** be code signed. Unsigned `official` apps won't be installable anymore.
- Apps which have been signed in a previous release **MUST** be code-signed in all future releases as well, otherwise the update will be refused.

How to Get Your App Signed

The following commands require that you have OpenSSL installed on your machine. Ensure that you keep all generated files to sign your application. The following examples will assume that you are trying to sign an application named "contacts".

1. Generate a private key and CSR: `openssl req -nodes -newkey rsa:2048 -keyout contacts.key -out contacts.csr -subj "/CN=contacts"`. Replace "contacts" with your application identifier.
2. Post the CSR at https://github.com/nextcloud/app-certificate-requests, and configure your GitHub account to show your mail address in your profile. Nextcloud might ask you for further information to verify that you're the legitimate owner of the application. Make sure to keep the private key file (`contacts.key`) secret and not disclose it to any third-parties.
3. Nextcloud will provide you with the signed certificate.
4. Run `./occ integrity:sign-app` to sign your application, and specify your private and the public key as well as the path to the application. A valid example looks like: `./occ integrity:sign-app --privateKey=/Users/lukasreschke/contacts.key --certificate=/Users/lukasreschke/CA/contacts.crt --path=/Users/lukasreschke/Programmi`

The occ tool will store a `signature.json` file within the `appinfo` folder of your application. Then compress the application folder and upload it to apps.owncloud.com. Be aware that doing any changes to the application after it has been signed requires another signing. So if you do not want to have some files shipped remove them before running the signing command.

In case you lose your certificate please submit a new CSR as described above and mention that you have lost the previous one. Nextcloud will revoke the old certificate.

If you maintain an app together with multiple people it is recommended to designate a release manager responsible for the signing process as well as the uploading to apps.owncloud.com. If there are cases where this is not feasible and multiple certificates are required Nextcloud can create them on a case by case basis. We do not recommend developers to share their private key.

Errors

The following errors can be encountered when trying to verify a code signature. For information about how to get access to those results please refer to the Issues section of the Nextcloud Server Administration manual.

- `INVALID_HASH`
  - The file has a different hash than specified within `signature.json`. This usually happens when the file has been modified after writing the signature data.
- `MISSING_FILE`
  - The file cannot be found but has been specified within `signature.json`. Either a required file has been left out, or `signature.json` needs to be edited.
- `EXTRA_FILE`
  - The file does not exist in `signature.json`. This usually happens when a file has been removed and `signature.json` has not been updated.
- `EXCEPTION`
  - Another exception has prevented the code verification. There are currently these following exceptions:

    `* Signature data not found.`

- The app has mandatory code signing enforced but no `signature.json` file has been found in its `appinfo` folder.

* `Certificate is not valid.`
  - The certificate has not been issued by the official Nextcloud Code Signing Root Authority.
* `Certificate is not valid for required scope. (Requested: %s, current: %s)`
  - The certificate is not valid for the defined application. Certificates are only valid for the defined app identifier and cannot be used for others.
* `Signature could not get verified.`
  - There was a problem with verifying the signature of `signature.json`.

App Development

Intro

Before you start, please check if there already is a similar app in the App Store or the GitHub organisation that you could contribute to. Also, feel free to communicate your idea and plans in the forum so other contributors might join in.

Then, please make sure you have set up a development environment:

- Development Environment

Before starting to write an app please read the security and coding guidelines:

- Security Guidelines
- Coding Style & General Guidelines

After this you can start with the tutorial

- Tutorial

Once you are ready for publishing, check out the app store process:

- App store publishing

For enhanced security it is also possible to sign your code:

- Code Signing

App development

Take a look at the changes in this version:

- Changelog

Create a new app:

- Create an app

Inner parts of an app:

- Navigation and Pre-App configuration
- App Metadata

- Classloader

Requests

How a request is being processed:

- Request lifecycle
- Routing
- Middleware
- Container
- Controllers | RESTful API

View

The app's presentation layer:

- Templates
- JavaScript
- CSS
- Translation
- Theming support

Storage

Create database tables, run Sql queries, store/retrieve configuration information and access the filesystem:

- Database Schema
- Database Access
- Configuration
- Filesystem

Authentication & Users

Creating, deleting, updating, searching, login and logout:

- Usermanagement

Writing a two-factor auth provider:

- Two-factor Providers

Hooks

Listen on events like user creation and execute code:

- Hooks

Background Jobs

Periodically run code in the background:

- Background Jobs (Cron)

Settings

An app can register both admin settings as well as personal settings:

- Settings

Logging

Log to the `data/nextcloud.log`:

- Logging

Testing

Write automated tests to ensure stability and ease maintenance:

- Testing

PHPDoc Class Documentation

Nextcloud class and function documentation:

- Nextcloud App API

Android Application Development

Nextcloud provides an official Nextcloud Android client, which gives its users access to their files on their Nextcloud. It also includes functionality like automatically uploading pictures and videos to Nextcloud.

For third party application developers, Nextcloud offers the Nextcloud Android library under the MIT license.

Android Nextcloud Client development

If you are interested in working on the Nextcloud android client, you can find the source code in github. The setup and process of contribution is documented here.

You might want to start with doing one or two starter issue to get into the code and note our General Contributor Guidelines

Nextcloud Android Library

This document will describe how to the use Nextcloud Android Library. The Nextcloud Android Library allows a developer to communicate with any Nextcloud server; among the features included are file synchronization, upload and download of files, delete rename files and folders, etc.

This library may be added to a project and seamlessly integrates any application with Nextcloud.

The tool needed is any IDE for Android preferred IDE at the moment is Android Studio.

Library Installation

Obtaining the library

The Nextcloud Android library may be obtained from the following Github repository:

https://github.com/nextcloud/android-library

Once obtained, this code should be compiled. The Github repository not only contains the library, but also a sample project, sample_client sample_client properties/android/librerias , which will assist in learning how to use the library.

Add the library to a project

There are different methods to add an external library to a project, we will describe two.

1. Add the library as a gradle dependency via jitpack
2. Add the library repo to your Android project as a git submodule

Add the library as a gradle dependency Simply open your:

```
build.gradle
```

and add the dependency:

```
compile 'com.github.nextcloud:android-library:<version>'
```

<version> refers to the exact version you would like to include in your application. This could be -SNAPSHOT for always using the latest code revision of the master branch. Alternatively you can also specifiy a version number which refers to a fixed release, e.g. 1.0.0. (compile 'com.github.nextcloud:android-library:1.0.0')

Add the library project to your project as a git submodule Basically get the code and compile it having it integrated via a git submodule

Go into your own apps directory on the command line and add the Nextcloud Android library as a submodule::
git submodule add https://github.com/nextcloud/android-library nextcloud-android-library

Import/Open your app in Android Studio and you are done. All the public classes and methods of the library will be available for your own app.

Examples

Init the library

Start using the library; it is needed to init the object mClient that will be in charge of keeping the communication with the server.

Code example

```
public class MainActivity extends Activity
                          implements  OnRemoteOperationListener,
                                      OnDatatransferProgressListener {
private OwnCloudClient mClient;
private Handler mHandler = new Handler();

...

public void onCreate(Bundle savedInstanceState) {

...

// Parse URI to the base URL of the Nextcloud server
Uri serverUri = Uri.parse(getString(R.string.server_base_url));

// Create client object to perform remote operations
mClient = OwnCloudClientFactory.createOwnCloudClient(
            serverUri,
            this,
            // Activity or Service context
            true);
```

Set credentials

Authentication on the app is possible by 3 different methods:

- Basic authentication, user name and password
- Bearer access token (oAuth2)
- Cookie (SAML-based single-sign-on)

Code example

```
package com.owncloud.android.lib.common;

public class OwnCloudClient extends HttpClient {
  ...
  // Set basic credentials
  client.setCredentials(
      NextcloudCredentialsFactory.newBasicCredentials(username, password)
  );
  // Set bearer access token
  client.setCredentials(
      NextcloudCredentialsFactory.newBearerCredentials(accessToken)
  );
  // Set SAML2 session token
  client.setCredentials(
      NextcloudCredentialsFactory.newSamlSsoCredentials(cookie)
```

```
  );
}
```

Create a folder

Create a new folder on the cloud server, the info needed to be sent is the path of the new folder.

Code example

```
private void startFolderCreation(String newFolderPath) {
  CreateRemoteFolderOperation createOperation = new CreateRemoteFolderOperation(newFolderPath, false)
  createOperation.execute( mClient , this , mHandler);
}

@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof CreateRemoteFolderOperation) {
    if (result.isSuccess()) {
    // do your stuff here
    }
  }
  // ...
}
```

Read folder

Get the content of an existing folder on the cloud server, the info needed to be sent is the path of the folder, in the example shown it has been asked the content of the root folder. As answer of this method, it will be received an array with all the files and folders stored in the selected folder.

Code example

```
private void startReadRootFolder() {
  ReadRemoteFolderOperation refreshOperation = new ReadRemoteFolderOperation(FileUtils.PATH_SEPARATOR
  // root folder
  refreshOperation.execute(mClient, this, mHandler);
}


@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof ReadRemoteFolderOperation) {
    if (result.isSuccess()) {
      List< RemoteFile > files = result.getData();
      // do your stuff here
    }
  }
  // ...
}
```

Read file

Get information related to a certain file or folder, information obtained is: `filePath`, `filename`, `isDirectory`, `size` and `date`.

Code example

```
private void startReadFileProperties(String filePath) {
  ReadRemoteFileOperation readOperation = new ReadRemoteFileOperation(filePath);
  readOperation.execute(mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof ReadRemoteFileOperation) {
    if (result.isSuccess()) {
      RemoteFile file = result.getData()[0];
      // do your stuff here
    }
  }
  // ...
}
```

Delete file or folder

Delete a file or folder on the cloud server. The info needed is the path of folder/file to be deleted.

Code example

```
private void startRemoveFile(String filePath) {
  RemoveRemoteFileOperation removeOperation = new RemoveRemoteFileOperation(remotePath);
  removeOperation.execute( mClient , this , mHandler);
}

@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof RemoveRemoteFileOperation) {
    if (result.isSuccess()) {
      // do your stuff here
    }
  }
  // ...
}
```

Download a file

Download an existing file on the cloud server. The info needed is path of the file on the server and targetDirectory, path where the file will be stored on the device.

Code example

```
private void startDownload(String filePath, File targetDirectory) {
  DownloadRemoteFileOperation downloadOperation = new DownloadRemoteFileOperation(filePath, targetDi
  downloadOperation.addDatatransferProgressListener(this);
```

```
  downloadOperation.execute( mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish( RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof DownloadRemoteFileOperation) {
    if (result.isSuccess()) {
      // do your stuff here
    }
  }
}

@Override
public void onTransferProgress( long progressRate, long totalTransferredSoFar, long totalToTransfer,
mHandler.post( new Runnable() {
  @Override
  public void run() {
    // do your UI updates about progress here
  }
});
}
```

Upload a file

Upload a new file to the cloud server. The info needed is fileToUpload, path where the file is stored on the device, remotePath, path where the file will be stored on the server and mimeType.

Code example

```
private void startUpload (File fileToUpload, String remotePath, String mimeType) {
  UploadRemoteFileOperation uploadOperation = new UploadRemoteFileOperation( fileToUpload.getAbsolute
  uploadOperation.addDatatransferProgressListener(this);
  uploadOperation.execute(mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof UploadRemoteFileOperation) {
    if (result.isSuccess()) {
      // do your stuff here
    }
  }
}

@Override
public void onTransferProgress(long progressRate, long totalTransferredSoFar, long totalToTransfer, S
  mHandler.post( new Runnable() {
    @Override
    public void run() {
      // do your UI updates about progress here
    }
  });
}
```

Move a file or folder

Move an exisintg file or folder to a different location in the Nextcloud server. Parameters needed are the path to the file or folder to move, and the new path desired for it. The parent folder of the new path must exist in the server.

When the parameter 'overwrite' is set to 'true', the file or folder is moved even if the new path is already used by a different file or folder. This one will be replaced by the former.

Code example

```
private void startFileMove(String filePath, String newFilePath, boolean overwrite) {
  MoveRemoteFileOperation moveOperation = new MoveRemoteFileOperation(filePath, newFilePath, overwrit
  moveOperation.execute( mClient , this , mHandler);
}

@Override
public void onRemoteOperationFinish(RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof MoveRemoteFileOperation) {
    if (result.isSuccess()) {
    // do your stuff here
        }
  }
  // ...
}
```

Read shared items by link

Get information about what files and folder are shared by link (the object mClient contains the information about the server url and account)

Code example

```
private void startAllSharesRetrieval() {
  GetRemoteSharesOperation getSharesOp = new GetRemoteSharesOperation();
  getSharesOp.execute( mClient , this , mHandler);
}

@Override
public void onRemoteOperationFinish( RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof GetRemoteSharesOperation) {
    if (result.isSuccess()) {
      ArrayList< OCShare > shares = new ArrayList< OCShare >();
      for (Object obj: result.getData()) {
        shares.add(( OCShare) obj);
      }
      // do your stuff here
    }
  }
}
```

Get the share resources for a given file or folder

Get information about what files and folder are shared by link on a certain folder. The info needed is filePath, path of the file/folder on the server, the Boolean variable, getReshares, come from the Sharing api, from the moment it is not in use within the Nextcloud Android library.

Code example

```
private void startSharesRetrievalForFileOrFolder(String filePath, boolean getReshares) {
  GeteRemoteSharesForFileOperation operation = new GetRemoteSharesForFileOperation(filePath, getResha
  operation.execute( mClient, this, mHandler);
}

private void startSharesRetrievalForFilesInFolder(String folderPath, boolean getReshares) {
  GetRemoteSharesForFileOperation operation = new GetRemoteSharesForFileOperation(folderPath, getResh
  operation.execute( mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish( RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof GetRemoteSharesForFileOperation) {
    if (result.isSuccess()) {
      ArrayList< OCShare > shares = new ArrayList< OCShare >();
      for (Object obj: result.getData()) {
        shares.add(( OCShare) obj);
      }
      // do your stuff here
   }
}
}
```

Share link of file or folder

Share a file or a folder from your cloud server by link.

The info needed is filePath, the path of the item that you want to share and Password, this comes from the Sharing api, from the moment it is not in use within the Nextcloud Android library.

Code example

```
private void startCreationOfPublicShareForFile(String filePath, String password) {
  CreateRemoteShareOperation operation = new CreateRemoteShareOperation(filePath, ShareType.PUBLIC_LI
  operation.execute( mClient , this , mHandler);
}

private void startCreationOfGroupShareForFile(String filePath, String groupId) {
  CreateRemoteShareOperation operation = new CreateRemoteShareOperation(filePath, ShareType.GROUP, gr
  operation.execute(mClient, this, mHandler);
}

private void startCreationOfUserShareForFile(String filePath, String userId) {
  CreateRemoteShareOperation operation = new CreateRemoteShareOperation(filePath, ShareType.USER, use
  operation.execute(mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish( RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof CreateRemoteShareOperation) {
    if (result.isSuccess()) {
      OCShare share = (OCShare) result.getData ().get(0);
      // do your stuff here
    }
  }
}
```

Delete a share resource

Stop sharing by link a file or a folder from your cloud server.

The info needed is the object OCShare that you want to stop sharing by link.

Code example

```
private void startShareRemoval(OCShare share) {
  RemoveRemoteShareOperation operation = new RemoveRemoteShareOperation((int) share.getIdRemoteShared
  operation.execute( mClient, this, mHandler);
}

@Override
public void onRemoteOperationFinish( RemoteOperation operation, RemoteOperationResult result) {
  if (operation instanceof RemoveRemoteShareOperation) {
    if (result.isSuccess()) {
    // do your stuff here
    }
  }
}
```

Tips

- Credentials must be set before calling any method
- Paths must not be on URL Encoding
- Correct path: `https://example.com/nextcloud/remote.php/dav/PopMusic`
- Wrong path: `https://example.com/nextcloud/remote.php/dav/Pop%20Music/`
- There are some forbidden characters to be used in folder and files names on the server, same on the Nextcloud Android Library “”,”/”,”<”,”>”,”:”,”“”,”|”,”?”,”*”
- Upload and download actions may be cancelled thanks to the objects uploadOperation.cancel(), downloadOperation.cancel()
- Unit tests, before launching unit tests you have to enter your account information (server url, user and password) on TestActivity.java

Translation

Make text translatable

In HTML or PHP wrap it like this `<?php p($l->t('This is some text'));?>` or this `<?php print_unescaped($l->t('This is some text'));?>` For the right date format use `<?php p($l->l('date', time()));?>`. Change the way dates are shown by editing /core/l10n/l10n-[lang].php To translate text in javascript use: `t('appname','text to translate');`

Note: `print_unescaped()` should be preferred only if you would like to display HTML code. Otherwise, using `p()` is strongly preferred to escape HTML characters against XSS attacks.

You shall never split sentences!

Reason:

Translators lose the context and they have no chance to possibly re-arrange words.

Example:

```
<?php p($l->t('Select file from')) . ' '; ?><a href='#' id="browselink"><?php p($l->t('local filesyst
```

Translators will translate:

- Select file from
- local filesystem
- ‘ or “
- cloud

Translating these individual strings results in `local filesystem` and `cloud` losing case. The two white spaces surrounding `or` will get lost while translating as well. For languages that have a different grammatical order it prevents the translators from reordering the sentence components.

Html on translation string:

Html tags in translation strings is ugly but usually translators can handle this.

What about variable in the strings?

If you need to add variables to the translation strings do it like this:

```
$l->t('%s is available. Get <a href="%s">more information</a>', array($data['versionstring'], $data[
```

Automated synchronization of translations

Multiple nightly jobs have been setup in order to synchronize translations - it’s a multi-step process: `perl l10n.pl read` will rescan all php and javascript files and generate the templates. The templates are pushed to Transifex (tx push -s). All translations are pulled from Transifex (tx pull -a). `perl l10n.pl write` will write the php files containing the translations. Finally the changes are pushed to git.

Please follow the steps below to add translation support to your app:

Create a folder `l10n`. Create the file `ignorelist` which can contain files which shall not be scanned during step 4. Edit `l10n/.tx/config` and copy/past a config section and adopt it by changing the app/folder name. Run `perl l10n.pl read` with l10n Add the newly created translation template (l10n/Templates/<appname>.pot) to git and commit the changes above. After the next nightly sync job a new resource will appear on Transifex and from now on every night the latest translations will arrive.

Translation sync jobs:

https://ci.owncloud.org/view/translation-sync/

Caution: information below is in general not needed!

Manual quick translation update:

```
cd l10n/ && perl l10n.pl read && tx push -s && tx pull -a && perl l10n.pl write && cd ..
```

The translation script requires Locale::PO, installable via `apt-get install liblocale-po-perl`

Configure transifex

```
tx init

for resource in calendar contacts core files media gallery settings
do
tx set --auto-local -r nextcloud.$resource "<lang>/$resource.po" --source-language=en \
 --source-file "templates/$resource.pot" --execute
done
```

Unit-Testing

PHP unit testing

Getting PHPUnit

Nextcloud uses PHPUnit >= 4.8 for unit testing.

To install it, either get it via your packagemanager:

```
sudo apt-get install phpunit
```

or install it manually:

```
wget https://phar.phpunit.de/phpunit.phar
chmod +x phpunit.phar
sudo mv phpunit.phar /usr/local/bin/phpunit
```

After the installation the ‘’phpunit” command is available:

```
phpunit --version
```

And you can update it using:

```
phpunit --self-update
```

You can find more information in the PHPUnit documentation: https://phpunit.de/manual/current/en/installation.html

Writing PHP unit tests

To get started, do the following:

- Create a directory called `tests` in the top level of your application
- Create a php file in the directory and `require_once` your class which you want to test.

Then you can simply run the created test with phpunit.

Note: If you use Nextcloud functions in your class under test (i.e: OC::getUser()) you'll need to bootstrap Nextcloud or use dependency injection.

Note: You'll most likely run your tests under a different user than the Web server. This might cause problems with your PHP settings (i.e: open_basedir) and requires you to adjust your configuration.

An example for a simple test would be:

`/srv/http/nextcloud/apps/myapp/tests/testaddtwo.php`

```
<?php
namespace OCA\Myapp\Tests;

class TestAddTwo extends \Test\TestCase {
    protected $testMe;

    protected function setUp() {
        parent::setUp();
        $this->testMe = new \OCA\Myapp\TestMe();
    }

    public function testAddTwo(){
          $this->assertEquals(5, $this->testMe->addTwo(3));
    }

}
```

`/srv/http/nextcloud/apps/myapp/lib/testme.php`

```
<?php
namespace OCA\Myapp;

class TestMe {
    public function addTwo($number){
        return $number + 2;
    }
}
```

In `/srv/http/nextcloud/apps/myapp/` you run the test with:

```
phpunit tests/testaddtwo.php
```

Make sure to extend the `\Test\TestCase` class with your test and always call the parent methods, when overwriting `setUp()`, `setUpBeforeClass()`, `tearDown()` or `tearDownAfterClass()` method from the TestCase. These methods set up important stuff and clean up the system after the test, so the next test can run without side effects, like remaining files and entries in the file cache, etc.

For more resources on PHPUnit visit: http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html

Bootstrapping Nextcloud

If you use Nextcloud functions or classes in your code, you'll need to make them available to your test by bootstrapping Nextcloud.

To do this, you'll need to provide the `--bootstrap` argument when running PHPUnit

`/srv/http/nextcloud`:

```
phpunit --bootstrap tests/bootstrap.php apps/myapp/tests/testsuite.php
```

If you run the test under a different user than your Web server, you'll have to adjust your php.ini and file rights.

`/etc/php/php.ini`:

```
open_basedir = none
```

`/srv/http/nextcloud`:

```
su -c "chmod a+r config/config.php"
su -c "chmod a+rx data/"
su -c "chmod a+w data/nextcloud.log"
```

Running unit tests for the Nextcloud core project

The core project provides a script that runs all the core unit tests using different database backends like sqlite, mysql, pgsql, oci (for Oracle):

```
./autotest.sh
```

To run tests only for sqlite:

```
./autotest.sh sqlite
```

To run a specific test suite (note that the test file path is relative to the "tests" directory):

```
./autotest.sh sqlite lib/share/share.php
```

Further Reading

- http://googletesting.blogspot.de/2008/08/by-miko-hevery-so-you-decided-to.html
- http://www.phpunit.de/manual/current/en/writing-tests-for-phpunit.html
- http://www.youtube.com/watch?v=4E4672CS58Q&feature=bf_prev&list=PLBDAB2BA83BB6588E
- Clean Code: A Handbook of Agile Software Craftsmanship (Robert C. Martin)

JavaScript unit testing for core

JavaScript Unit testing for **core** and **core apps** is done using the Karma test runner with Jasmine.

Installing Node JS

To run the JavaScript unit tests you will need to install **Node JS**.

You can get it here: http://nodejs.org/

After that you will need to setup the **Karma** test environment. The easiest way to do this is to run the automatic test script first, see next section.

Running all tests

To run all tests, just run:

```
./autotest-js.sh
```

This will also automatically set up your test environment.

Debugging tests in the browser

To debug tests in the browser, you need to run **Karma** in browser mode:

```
karma start tests/karma.config.js
```

From there, open the URL http://localhost:9876 in a web browser.

On that page, click on the "Debug" button.

An empty page will appear, from which you must open the browser console (F12 in Firefox/Chrome).

Every time you reload the page, the unit tests will be relaunched and will output the results in the browser console.

Unit test paths

JavaScript unit test examples can be found in `apps/files/tests/js/`

Unit tests for the core app JavaScript code can be found in `core/js/tests/specs`

Documentation

Here are some useful links about how to write unit tests with Jasmine and Sinon:

- Karma test runner: http://karma-runner.github.io
- Jasmine: http://pivotal.github.io/jasmine
- Sinon (for mocking and stubbing): http://sinonjs.org/

Theming Nextcloud

Themes can be used to customize the look and feel of Nextcloud. Themes can relate to the following topics of Nextcloud:

- Theming the web-frontend
- Theming the desktop client

This documentation contains only the Web-frontend adaptations so far.

Getting started

A good idea getting starting with a dynamically created website is to inspect it with **web developer tools**, that are found in almost any browser. They show the generated HTML and the CSS Code, that the client/browser is receiving: With this facts you can easily determine, where the following object-related attributes for the phenomenons are settled:

- place
- colour
- links
- graphics

The next thing you should do, before starting any changes is: Make a backup of your current theme(s) e.g.:

- cd . . . /nextcloud/themes
- cp -r example mytheme

Creating and activating a new theme

There are two basic ways of creating new themings:

- Doing all new from scratch
- Starting from an existing theme or the example theme and doing everything step by step and more experimentally

Depending on how you created your new theme it will be necessary to

- put a new theme into the /themes -folder. The theme can be activated by putting `'theme' => 'MyTheme'`, into the `/config/config.php` file.
- make your changes in the `/themes/MyTheme` -folder
- make sure that the theming app is disabled

Structure

The folder structure of a theme is exactly the same as the main Nextcloud structure. You can override js files, images, translations and templates with own versions. CSS files are loaded additionally to the default files so you can override CSS properties. CSS files and the standard pictures that are used reside for example in /nextcloud/core/ and /nextcloud/settings/ in these sub folders:

- css = style sheets
- js = JavaScripts
- img = images
- l10n = translation files
- templates = php and html template files

Notes for Updates

It is not recommended to the user to perform adaptations inside the folder `/themes/example` because files inside this folder might get replaced during the next Nextcloud update process.

During an update, files might get changed within the core and settings folders. This could result in problems because your template files will not 'know' about these changes and therefore must be manually merged with the updated core file or simply be deleted (or renamed for a test).

For example if `/settings/templates/apps.php` gets updated by a new Nextcloud version, and you have a `/themes/MyTheme/settings/templates/apps.php` in your template, you must merge the changes that where made within the update with the ones you did in your template.

But this is unlikely and will be mentioned in the Nextcloud release notes if it occurs.

How to change images and the logo

A new logo which you may want to insert can be added as follows:

Figure out the path of the old logo

Replace the old picture, which position you found out as described under 1.3. by adding an extension in case you want to re-use it later.

Creating an own logo

If you want to do a quick exchange like (1) it's important to know the size of the picture before you start creating an own logo:

- Go to the place in the filesystem, that has been shown by the web developer tool/s
- You can look up sizing in most cases via the file properties inside your file-manager
- Create an own picture/logo with the same size then

The (main) pictures, that can be found inside Nextcloud standard theming are the following:

- The logo at the login-page above the credentials-box: . . . /nextcloud/themes/default/core/img/logo.svg
- The logo, that's always in the left upper corner after login: . . . /nextcloud/themes/default/core/img/logo-wide.svg

Inserting your new logo

Inserting a new logo into an existing theme is as simple as replacing the old logo with the new (generated) one. You can use: scalable vector graphics (.svg) or common graphics formats for the internet such as portable network graphics (.png) or .jepg Just insert the new created picture by using the unchanged name of the old picture.

The app icons can also be overwritten in a theme. To change for example the app icon of the activity app you need to overwrite it by saving the new image to . . . /nextcloud/themes/default/apps/activity/img/activity.svg

Changing favicon

For compatibility with older browsers, favicon (the image that appears in your browser tab) uses .../nextcloud/core/img/favicon.ico.

To customize favicon for MyTheme:

- Create a version of your logo in .ico format
- Store your custom favicon as .../nextcloud/themes/MyTheme/core/img/favicon.ico
- Include .../nextcloud/themes/MyTheme/core/img/favicon.svg and favicon.png to cover any future updates to favicon handling.

Changing the default colours

With a web-developer tool like Mozilla-Inspector, you also get easily displayed the color of the background you clicked on. On the top of the login page you can see a case- distinguished setting for different browsers:

```
/* HEADERS */
...
body-login {
  background: #1d2d42; /* Old browsers */
  background: -moz-linear-gradient(top, #33537a 0%, #1d2d42  100%); /* FF3.6+ */
  background: -webkit-gradient(linear, left top, left bottom, color-stop(0%,#F1B3A4), color-stop(100%
  background: -webkit-linear-gradient(top, #33537a 0%,#1d2d42 100%); /* Chrome10+,Safari5.1+ */
  background: -o-linear-gradient(top, #33537a 0%,#1d2d42 100%); /* Opera11.10+ */
  background: -ms-linear-gradient(top, #33537a 0%,#1d2d42 100%); /* IE10+ */
  background: linear-gradient(top, #33537a 0%,#1d2d42 100%); /* W3C */
}
```

The different background-assignments indicate the headers for a lot of different browser types. What you most likely want to do is change the #35537a (lighter blue) and #ld2d42 (dark blue) color to the colours of our choice. In some older and other browsers, there is just one color, but in the rest showing gradients is possible. The login page background is a horizontal gradient. The first hex number, #35537a, is the top color of the gradient at the login screen. The second hex number, #ld2d42 is the bottom color of the gradient at the login screen. The gradient in top of the normal view after login is also defined by these CSS-settings, so that they take effect in logged in situation as well. Change these colors to the hex color of your choice: As usual:

- the first two figures give the intensity of the red channel,
- the second two give the green intensity and the
- third pair gives the blue value.

Save your CSS-file and refresh to see the new login screen. The other major color scheme is the blue header bar on the main navigation page once you log in to Nextcloud. This color we will change with the above as well. Save the file and refresh the browser for the changes to take effect.

How to change translations

You can override the translation of single strings within your theme. Simply create the same folder structure within your theme folder for the language file you want to override. Only the changed strings need to be added to that file for all other terms the shipped translation will be used.

If you want to override the translation of the term “Download” within the `files` app for the language `de` you need to create the file `themes/THEME_NAME/apps/files/l10n/de.js` and put the following code in:

```
OC.L10N.register(
  "files",
  {
    "Download" : "Herunterladen"
  },
  "nplurals=2; plural=(n != 1);"
);
```

Additionally you need to create another file `themes/THEME_NAME/apps/files/l10n/de.json` with the same translations that look like this:

```
{
  "translations": {
    "Download" : "Herunterladen"
  },
  "pluralForm" :"nplurals=2; plural=(n != 1);"
}
```

Both files (`.js` and `.json`) are needed with the same translations, because the first is needed to enable translations in the JavaScript code and the second one is read by the PHP code and provides the data for translated terms in there.

How to change names, slogans and URLs

The Nextcloud theming allows a lot of the names that are shown on the web interface to be changed. It's also possible to change the URLs to the documentation or the Android/iOS apps.

This can be done with a file named `defaults.php` within the root of the theme. You can find it in the example theme (*/themes/example/defaults.php*). In there you need to specify a class named `OC_Theme` and need to implement the methods you want to overwrite:

```
class OC_Theme {
  public function getAndroidClientUrl() {
    return 'https://play.google.com/store/apps/details?id=com.nextcloud.client';
  }

  public function getName() {
    return 'Nextcloud';
  }
}
```

Each method should return a string. Following methods are available:

- `getAndroidClientUrl`
- `getBaseUrl`
- `getDocBaseUrl`
- `getEntity`
- `getName`
- `getHTMLName`
- `getiOSClientUrl`
- `getiTunesAppId`
- `getLogoClaim`

- `getLongFooter`
- `getMailHeaderColor`
- `getSyncClientUrl`
- `getTitle`
- `getShortFooter`
- `getSlogan`

Note: Only these methods are available in the templates, because we internally wrap around hardcoded method names.

One exception is the method `buildDocLinkToKey` which gets passed in a key as first parameter. For core we do something like this to build the documentation link:

```
public function buildDocLinkToKey($key) {
  return $this->getDocBaseUrl() . '/server/9.0/go.php?to=' . $key;
}
```

Testing the new theme out

There are different options for doing so:

- If you're using a tool like the Inspector tools inside Mozilla, you can test out the CSS-Styles immediately inside the css-attributes, while looking at them.
- If you have a developing/testing server as described in 1. you can test out the effects in a real environment permanently.

App config

```
<?php

$CONFIG = array(
/* Flag to indicate Nextcloud is successfully installed (true = installed) */
"installed" => false,

/* Type of database, can be sqlite, mysql or pgsql */
"dbtype" => "sqlite",

/* Name of the Nextcloud database */
"dbname" => "nextcloud",

/* User to access the Nextcloud database */
"dbuser" => "",

/* Password to access the Nextcloud database */
"dbpassword" => "",

/* Host running the Nextcloud database */
"dbhost" => "",
```

```
/* Prefix for the Nextcloud tables in the database */
"dbtableprefix" => "",

/* Define the salt used to hash the user passwords. All your user passwords are lost if you lose this
"passwordsalt" => "",

/* Force use of HTTPS connection (true = use HTTPS) */
"forcessl" => false,

/* Theme to use for Nextcloud */
"theme" => "",

/* Path to the 3rdparty directory */
"3rdpartyroot" => "",

/* URL to the 3rdparty directory, as seen by the browser */
"3rdpartyurl" => "",

/* Default app to load on login */
"defaultapp" => "files",

/* Enable the help menu item in the settings */
"knowledgebaseenabled" => true,

/* Enable installing apps from the appstore */
"appstoreenabled" => true,

/* URL of the appstore to use, server should understand OCS */
"appstoreurl" => "https://api.owncloud.com/v1",

/* Mode to use for sending mail, can be sendmail, smtp, qmail or php, see PHPMailer docs */
"mail_smtpmode" => "sendmail",

/* Host to use for sending mail, depends on mail_smtpmode if this is used */
"mail_smtphost" => "127.0.0.1",

/* authentication needed to send mail, depends on mail_smtpmode if this is used
 * (false = disable authentication)
 */
"mail_smtpauth" => false,

/* Username to use for sendmail mail, depends on mail_smtpauth if this is used */
"mail_smtpname" => "",

/* Password to use for sendmail mail, depends on mail_smtpauth if this is used */
"mail_smtppassword" => "",

/* Check 3rdparty apps for malicious code fragments */
"appcodechecker" => "",

/* Check if Nextcloud is up to date */
"updatechecker" => true,

/* Place to log to, can be owncloud and syslog (owncloud is log menu item in admin menu) */
"log_type" => "owncloud",

/* File for the owncloud logger to log to, (default is nextcloud.log in the data dir */
"logfile" => "",
```

```
/* Loglevel to start logging at. 0=DEBUG, 1=INFO, 2=WARN, 3=ERROR (default is WARN) */
"loglevel" => "",

/* Lifetime of the remember login cookie, default is 15 days */
"remember_login_cookie_lifetime" => 60*60*24*15,

/* The directory where the user data is stored, default to data in the owncloud
 * directory. The sqlite database is also stored here, when sqlite is used.
 */
// "datadirectory" => "",

"apps_paths" => array(

/* Set an array of path for your apps directories
 key 'path' is for the fs path and the key 'url' is for the http path to your
 applications paths. 'writable' indicate if the user can install apps in this folder.
 You must have at least 1 app folder writable or you must set the parameter : appstoreenabled to fals
*/
        array(
                'path'=> '/var/www/nextcloud/apps',
                'url' => '/apps',
                'writable' => true,
  ),
 ),
);
```

Using alternative app directories

Nextcloud can be set to use a custom app directory in /config/config.php. Customize the following code and add it to your config file:

```
'apps_paths' =>
      array (
              0 =>
              array (
                      'path' => OC::$SERVERROOT.'/apps',
                      'url' => '/apps',
                      'writable' => true,
              ),
              1 =>
              array (
                      'path' => OC::$SERVERROOT.'/apps2',
                      'url' => '/apps2',
                      'writable' => false,
              ),
      ),
```

Nextcloud will use the first app directory which it finds in the array with 'writable' set to true.

External API

Introduction

The external API inside Nextcloud allows third party developers to access data provided by Nextcloud apps. Nextcloud follows the OCS v1.7 specification (draft).

Usage

Registering Methods

Methods are registered inside the `appinfo/routes.php` using `OCP\API`

```
<?php

\OCP\API::register(
    'get',
    '/apps/yourapp/url',
    function($urlParameters) {
      return new \OC_OCS_Result($data);
    },
    'yourapp',
    \OC_API::ADMIN_AUTH
);
```

Returning Data

Once the API backend has matched your URL, your callable function as defined in **$action** will be executed. This method is passed as array of parameters that you defined in **$url**. To return data back the the client, you should return an instance of `OC_OCS_Result`. The API backend will then use this to construct the XML or JSON response.

Authentication & Basics

Because REST is stateless you have to send user and password each time you access the API. Therefore running Nextcloud **with SSL is highly recommended** otherwise **everyone in your network can log your credentials**:

```
https://user:password@example.com/ocs/v1.php/apps/yourapp
```

Output

The output defaults to XML. If you want to get JSON append this to the URL:

```
?format=json
```

Output from the application is wrapped inside a **data** element:

XML:

```
<?xml version="1.0"?>
<ocs>
 <meta>
  <status>ok</status>
  <statuscode>100</statuscode>
```

```
  <message/>
 </meta>
 <data>
   <!-- data here -->
 </data>
</ocs>
```

JSON:

```
{
  "ocs": {
    "meta": {
      "status": "ok",
      "statuscode": 100,
      "message": null
    },
    "data": {
      // data here
    }
  }
}
```

Statuscodes

The statuscode can be any of the following numbers:

- **100** - successful
- **996** - server error
- **997** - not authorized
- **998** - not found
- **999** - unknown error

OCS Share API

The OCS Share API allows you to access the sharing API from outside over pre-defined OCS calls.

The base URL for all calls to the share API is: *<nextcloud_base_url>/ocs/v1.php/apps/files_sharing/api/v1*

All calls to OCS endpoints require the `OCS-APIRequest` header to be set to `true`.

Local Shares

Get All Shares

Get all shares from the user.

- Syntax: /shares
- Method: GET
- Result: XML with all shares

Statuscodes:

- 100 - successful
- 404 - couldn't fetch shares

Get Shares from a specific file or folder

Get all shares from a given file/folder.

- Syntax: /shares
- Method: GET
- URL Arguments: path - (string) path to file/folder
- URL Arguments: reshares - (boolean) returns not only the shares from the current user but all shares from the given file.
- URL Arguments: subfiles - (boolean) returns all shares within a folder, given that *path* defines a folder
- Mandatory fields: path
- Result: XML with the shares

Statuscodes

- 100 - successful
- 400 - not a directory (if the 'subfile' argument was used)
- 404 - file doesn't exist

Get information about a known Share

Get information about a given share.

- Syntax: /shares/<*share_id*>
- Method: GET
- Arguments: share_id - (int) share ID
- Result: XML with the share information

Statuscodes:

- 100 - successful
- 404 - share doesn't exist

Create a new Share

Share a file/folder with a user/group or as public link.

- Syntax: /shares
- Method: POST
- POST Arguments: path - (string) path to the file/folder which should be shared
- POST Arguments: shareType - (int) 0 = user; 1 = group; 3 = public link; 6 = federated cloud share
- POST Arguments: shareWith - (string) user / group id with which the file should be shared
- POST Arguments: publicUpload - (boolean) allow public upload to a public shared folder (true/false)

- POST Arguments: password - (string) password to protect public link Share with
- POST Arguments: permissions - (int) 1 = read; 2 = update; 4 = create; 8 = delete; 16 = share; 31 = all (default: 31, for public shares: 1)
- Mandatory fields: shareType, path and shareWith for shareType 0 or 1.
- Result: XML containing the share ID (int) of the newly created share

Statuscodes:

- 100 - successful
- 400 - unknown share type
- 403 - public upload was disabled by the admin
- 404 - file couldn't be shared

Delete Share

Remove the given share.

- Syntax: /shares/<*share_id*>
- Method: DELETE
- Arguments: share_id - (int) share ID

Statuscodes:

- 100 - successful
- 404 - file couldn't be deleted

Update Share

Update a given share. Only one value can be updated per request.

- Syntax: /shares/<*share_id*>
- Method: PUT
- Arguments: share_id - (int) share ID
- PUT Arguments: permissions - (int) update permissions (see "Create share" above)
- PUT Arguments: password - (string) updated password for public link Share
- PUT Arguments: publicUpload - (boolean) enable (true) /disable (false) public upload for public shares.
- PUT Arguments: expireDate - (string) set a expire date for public link shares. This argument expects a well formatted date string, e.g. 'YYYY-MM-DD'

Note: Only one of the update parameters can be specified at once.

Statuscodes:

- 100 - successful
- 400 - wrong or no update parameter given
- 403 - public upload disabled by the admin

- 404 - couldn't update share

Federated Cloud Shares

Both the sending and the receiving instance need to have federated cloud sharing enabled and configured. See Configuring Federated Cloud Sharing. .. TODO ON RELEASE: Update version number above on release

Create a new Federated Cloud Share

Creating a federated cloud share can be done via the local share endpoint, using (int) 6 as a shareType and the Federated Cloud ID of the share recipient as shareWith. See *Create a new Share* for more information.

List accepted Federated Cloud Shares

Get all federated cloud shares the user has accepted.

- Syntax: /remote_shares
- Method: GET
- Result: XML with all accepted federated cloud shares

Statuscodes:

- 100 - successful

Get information about a known Federated Cloud Share

Get information about a given received federated cloud that was sent from a remote instance.

- Syntax: /remote_shares/<*share_id*>
- Method: GET
- Arguments: share_id - (int) share ID as listed in the id field in the `remote_shares` list
- Result: XML with the share information

Statuscodes:

- 100 - successful
- 404 - share doesn't exist

Delete an accepted Federated Cloud Share

Locally delete a received federated cloud share that was sent from a remote instance.

- Syntax: /remote_shares/<*share_id*>
- Method: DELETE
- Arguments: share_id - (int) share ID as listed in the id field in the `remote_shares` list
- Result: XML with the share information

Statuscodes:

- 100 - successful

- 404 - share doesn't exist

List pending Federated Cloud Shares

Get all pending federated cloud shares the user has received.

- Syntax: /remote_shares/pending
- Method: GET
- Result: XML with all pending federated cloud shares

Statuscodes:

- 100 - successful

Accept a pending Federated Cloud Share

Locally accept a received federated cloud share that was sent from a remote instance.

- Syntax: /remote_shares/pending/<*share_id*>
- Method: POST
- Arguments: share_id - (int) share ID as listed in the id field in the `remote_shares/pending` list
- Result: XML with the share information

Statuscodes:

- 100 - successful
- 404 - share doesn't exist

Decline a pending Federated Cloud Share

Locally decline a received federated cloud share that was sent from a remote instance.

- Syntax: /remote_shares/pending/<*share_id*>
- Method: DELETE
- Arguments: share_id - (int) share ID as listed in the id field in the `remote_shares/pending` list
- Result: XML with the share information

Statuscodes:

- 100 - successful
- 404 - share doesn't exist

Core Development

Intro

Please make sure you have set up a development environment:

- Development Environment

Core related docs

- Translation
- Unit-Testing
- Theming Nextcloud
- App config
- OCS Share API
- External API

Bugtracker

Code Reviews on GitHub

> Given enough eyeballs, all bugs are shallow
>
> —Linus' Law

Introduction

In order to increase the code quality within Nextcloud, developers are requested to perform code reviews. As we are now heavily using the GitHub platform these code review shall take place on GitHub as well.

Precondition

From now on no direct commits/pushes to master or any of the stable branches are allowed in general. **Every code** change - **even one liners** - have to be reviewed!

How will it work?

1. A developer will submit his changes on GitHub via a pull request (PR). GitHub:help - using pull requests
2. Within the pull request the developer could already name other developers (using @GitHubusername) and ask them for review.
3. Using Labels section on the right side, they add *"3 - To review"* label if the patch is complete. If they have no permission to do that, other developers may add this Label in case PR author had indicated.
4. Other developers (either named or at free will) have a look at the changes and are welcome to write comments within the comment field.
5. In case the reviewer is okay with the changes and thinks all his comments and suggestions have been take into account a :+1 on the comment will signal a positive review.
6. Before a pull request will be merged into master or the corresponding branch at least 2 reviewers need to give :+1 score.
7. Our continuous integration server will give an additional indicator for the quality of the pull request.

Examples

Read our coding guidelines for information on what a good pull request and good Nextcloud code looks like.

These are two examples that are considered to be good examples of how pull requests should be handled

- https://github.com/owncloud/core/pull/121
- https://github.com/owncloud/core/pull/146

Questions?

Feel free to drop a line on the forums or join us on IRC.

Development process

This chapter contains a lot of information about the development process the Nextcloud community tries to follow, so please take your time to digest all the information. In any case remember this page as the documentation on how it should be done. Nothing here is set in stone, so if you think something should be changed please discuss it on the forums.

The Labels

The following list shows what the labels mean in the life-cycle and will hopefully help you decide how to label an issue.

Backlog

Why do we have it? To keep us focused on finishing issues that we started, new issues will be hidden in this column. In huboard you can see the list of things that we could think about by clicking the small arrow in the top left corner of the concept column header.

What does a developer think? "Maybe later."

When can I pull? Since this is the bucket for whatever might be done you should only pick issues from the backlog when there is no other issue that you can work on. It is more important to finish an issue currently on the Kanban board than to pull a new one into the flow because only released issues have a value to our users!

Who is Assigned? Either a maintainer feels directly responsible for the issue and assigns himself or the gatekeeper (the guys having a look at unassigned bugs) will try to determine the responsible developer.

Concept

Why do we have it? Our think before you act phase serves two purposes. A Bug is in the concept phase while we are trying to figure out why something is broken (analysis). An Enhancement is in the concept phase until we have decided how to implement it (design).

What does a developer think? "I'll write a Scenario for our BDD in Gherkin and post it as a comment. I can always look at the existing ones to get an inspiration how to phrase them as "Given ... when ... then ...""

When can I pull? As long as you think and discuss on how to implement an enhancement or how to solve a bug you should leave the concept label assigned. Two things should be documented in a comment to the issue before moving it to the "To develop" step:

- At least one Scenario – written in Gherkin – that tells you and the tester when the issue is ready to be released.
- A concept describing the planned implementation. This can be as simple as a "this just needs changes to the login screen css" or so complex that you link to a blog entry somewhere else.

Who is Assigned? The maintainer that feels responsible for the issue.

To Develop

Why do we have it? Now that we have a plan, any developer can pick an issue from this column and start implementing it. If the issue is also marked with Junior Job this might be a good starting point for new developers.

What does a developer think? "Nice! I can safely implement it that way because more than one person has put his brain to the task of coming up with a good solution. Here! Me! I'll do it!"

When can I pull? If you feel like diving into the code and getting your hands dirty you should look for issues with this label. In the comments, there should be a gherkin scenario to tell you when you are done and a concept describing how to implement it. Before you start move the issue to the "Developing" step by assigning the "4 – Developing" label.

Who is Assigned? No one. Especially not if you are working on something else!

Developing

Why do we have it? This is where the magic happens. If it's a Bug the fix will be submitted as a PULL REQUEST to the master or corresponding stable branch. If its an Enhancement code will be committed to a feature branch.

What does a developer think? "You know, I'm at it. By the way, I'll also write unit tests. When I'm done I'll push the issue with a commit containing "push GH-#" where # is the issue number. If I have an idea of who should review it I can also notify them with @githubusername"

When can I pull? As long as you are writing code for the issue or if any unit test fails you should leave the "4 – Developing" label assigned. Two things should have been implemented before moving the issue to the "To review" step:

- The enhancement or bug in question
- Unit tests for the changed and added code.

Who is Assigned? The most active developer should assign himself.

To Review

Why do we have it? Instead of directly committing to master we agree that **a second set of eyes will spot bugs** and increase our code quality and give us an opportunity to learn from each other. See also our Code Review Documentation

What does a developer think? "I'll check the Scenario described earlier works as expected. If necessary I'll update the related Gherkin Scenarios. Drone will test the scenario on all kinds of platforms, Web server and database combinations with cucumber."

When can I pull? If you feel like making sure an issue works as expected you should look for issues with this label. In the comments you should find a gherkin scenario that can be used as a checklist for what to try. Before you start move the issue to the "Reviewing" step by assigning the "6 – Reviewing" label.

Who is Assigned? No one. Especially not if you are working on something else!

Reviewing

Why do we have it? With the Gherkin Scenario from the Concept Phase reviewers have a checklist to test if a Bug has been solved and if an Enhancement works as expected. **The most eager reviewer we have is Drone**. When it comes to testing he soldiers on going through the different combinations of platform, Web server and database.

What does a developer think? "Damn! If I had written the Gherkin Scenarios and Cucumber Step Definitions I could leave the task of testing this on the different combinations of platform, Web server and database to Drone. I'll miss something when doing this manually.*

When can I pull? As long as you are reviewing the issue the you should leave the "6 – Reviewing" label assigned. Before moving the issue to the "To review" step the issue should have been resolved, meaning that not only the issue has been implemented but also no other functionality has been broken.

Who is Assigned? The most active reviewer should assign himself.

To Release

Why do we have it? This is a list of issues that will make it into the next release. It serves as a source for the changelog, as well as a reminder of the work we can already be proud of.

What does a developer think? "Look at all the shiny things we will release with the next version of Nextcloud!"

When can I pull? This is the last step of the Kanban board. When the Release finally happens the issue will be closed and removed from the board.

Who is Assigned? No one.

While we stated before that said that we push issues to the next column, we can of course move the item back and forth arbitrarily. Basically you can drag the issue around in the huboard or just change the label when viewing the issue in the GitHub.

Reviewing considered impossible?

How can you possibly review an issue when it requires you to test various combinations of browsers, platforms, databases and maybe even app combinations? Well, you can't. But you can write a gherkin scenario that can be used to write an automated test that is executed by Drone on every commit to the main repositories. If for some reason Drone cannot be used for the review you will find yourself in the very uncomfortable situation where you release half tested code that will hopefully not eat user data. Seriously! Write gherkin scenarios!

Other Labels

Priority Labels

- Panic should be used with caution. It is reserved for Bugs that would result in the loss of files or other user data. An Enhancement marked as Panic is expected by Nextcloud users for the next release. In either case an open Panic issue will prevent a release.
- Attention is not as hard as Panic. But we really want this in the next release and will dedicate more effort for it. But if we think the issue is not ready for the next release we will postpone it to the next one.
- Regression is something that worked in a previous release but is now not working as expected or missing. If a certain functionality is up for code refactoring, the developer should describe all possible use cases as a Gherkin scenarios beforehand, so that any scenarios that isn't implemented before the required milestone can be marked as a regression. If a regression is found after a release, the reporter – or the developer triaging the issue – should

describe the functionality as a Gherkin scenario and either fix it or assign it to the developer in charge of that part.

App Labels

In the apps repository there are labels like `app:gallery` and `app:calendar`. The `app:` prefix is used to allow developers to filter issues related to a specific app.

Resolution Status

- Fixed – Should be assigned to issues in to Release
- Won't fix – Reason is given as a comment
- Duplicate – Corresponding bug is given in a comment (using #guthubissuenumber)

Misc Labels

- Needs info – Either from a developer or the bug reporter. This is nearly as severe as Panic, because no further action can be taken
- L18n – A translation issue go see our transifex
- Junior Job – The issue is considered a good starting point to get involved in Nextcloud development

Milestones equal Releases

Releases are planned via milestones which contain all the Enhancements and Bugs that we plan to release when the Deadline is met. When the Deadline approaches we will push new Enhancement request and less important bugs to the next milestone. This way a milestone will upon release contain all the issues that make up the changelog for the release. Furthermore, huboard allows us to filter the Kanban board by Milestone, making it especially easy to focus on the current Release.

Nextcloud Bug Triaging

Bug Triaging is the process of checking bug reports to see if they are still valid (the problem might be solved since the bug was reported), reproducing them when possible (to make sure it really is an Nextcloud issue and not a configuration problem) and in general making sure the bug is useful for a developer who wants to fix it. If the bug is not useful and can't be augmented by the original reporter or the triaging contributor, it has to be closed.

Why do you want to join

Helping to bring the number of issues down makes it easier for developers to spend their time productively and bug triagers thus **contribute greatly to Nextcloud development**! Triaging a bug doesn't take long so the work comes in small chunks and you don't need many skills, just some patience and sometimes perseverance.

Bug triagers who contribute significantly should ask to be listed as an active contributor on the nextcloud.com page!

How do you triage bugs

The process of checking, reproducing and closing invalid issues is called 'bug triaging'. Issues can be divided in one of three kinds:

1. Bugs or feature requests which come with all needed information to allow a developer to fix or work on them
2. Incomplete or duplicate bug reports or feature requests
3. Irrelevant or wrong bug reports or feature requests

The job of a bug triager is to identify the One's for developers to look at, help remove, merge or improve any Two to a One and dismiss Three's in a friendly and emphatic way.

Triaging follows these steps:

- Find an issue somebody should look at
- Be that somebody and see if the issue content is useful for a developer
- Reply and close, ask a question, add information or a label.
- Find the next bug-to-be-dealt-with and repeat!

General considerations

- You need a github account to contribute to bug triaging.
- If you are not familiar with the github issue tracker interface (which is used by Nextcloud to handle bug reports), you may find this guide useful.
- You will initially only be able to comment on issues. The ability to close issues or assign labels will be given liberally to those who have shown to be willing and able to contribute. Just ask on IRC!
- Read our bug reporting guidelines so you know what a good report should look like and where things belong. The issue template asks specifically for some information developers need to solve issues.
- It might even be fixed, sometimes! It can also be fruitful to contact the developers on irc. Tell them you're triaging bugs and share what problem you bumped into. Or just ask on the test-pilots mailing list.
- To ensure no two people are working on the same issue, we ask you to simply add a comment like "I am triaging this" in the issue you want to work on, and when done, before or after executing the triaging actions, note similarly that you're done.

  > To be able to tag and close issues, you need to have access to the repository. For the core and sync app repositories this also means having signed the contributor agreement. However, this isn't really needed for triaging as you can comment after you're done triaging and somebody else can execute those actions.

Finding bugs to triage

Github offers several search queries which can be useful to find a list of bugs which deserve a closer look:

- The bugs least recently commented on
- Least commented issues
- Bugs which need info

But there are more methods. For example, if you are a user of Nextcloud with a specific setup like using nginx as Web server or dropbox as storage, or using the encryption app, you could look for bugs with these keywords. You can then use your knowledge of your installation and your installation itself to see if bugs are (still) valid or reproduce them.

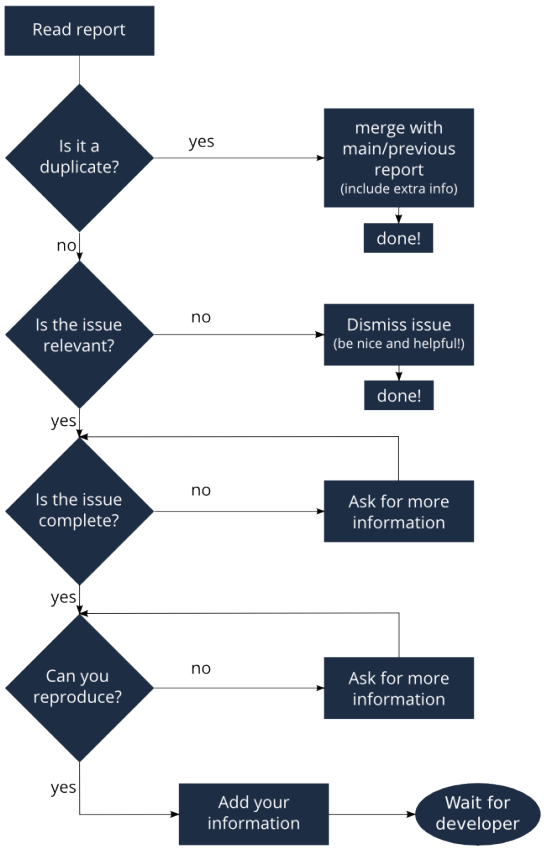
Once you have picked an issue, add a comment that you've started triaging:

> "I am triaging this bug"

Checking if the issue is useful

Much content from https://techbase.kde.org/Contribute/Bugsquad/Guide_To_BugTriaging

The goal of triaging is to have only useful bug reports for the developers. And you don't have to know much to be able to judge at least some bug reports to be less than useful. There are duplications, incomplete reports and so on. Here is the work flow for each bug:



Let's go over each step.

Finding duplicates

To find duplicates, the search tool in github is your first stop. In this screen you can easily search for a few keywords from the bug report. If you find other bugs with the same content, decide what the best bug report is (often the oldest or the one where one or more developers have already started to engage and discuss the problem). That is the 'master' bug report, you can now close the other one (or comment that it can be closed as duplicate).

If the bug report you were reviewing contains additional information, you can add that information to the 'master' bug report in a comment. Mention this bug report (using #<bug report number>) so a developer can look up the original, closed, report and perhaps ask the initial reporter there for additional information.

If you can't find anything, look in closed bug reports. The problem might be solved already and be listed there! Of course, these other bug reports might be closed as duplicates of the one you are looking at now - if you can't find one that is solved nor can find any duplicates, you can move on to the next step. If you are unsure, just add a comment: "might be a duplicate of #<bug nr here>" will usually suffice.

When the issue is a feature request, you can be helpful in the same way: merge related requests by adding information of one to the other and closing the first.

Note: Be polite: when you need to request information or feedback be clear and polite, and you will get more information in less time. Think about how you'd like to be treated, were you to report a bug!

Note: You can answer more quickly and friendly using one of these templates.

Note: Often our github issue tracker is a place for discussions about solutions. Be friendly, inclusive and respect other people's position.

Determining relevance of issue

Not all issues are relevant for Nextcloud. Bugs can be due to a specific configuration or unsupported platforms. Raspberry Pi's suffer from SQLite time-outs, nginx has problems Apache doesn't and Microsoft Server with IIS is not well supported. While external issues are not always a reason to close a report, be sure that they are clear: does the user use the 'standard' platform? Ask for information if this is missing.

Last but not least, the problem might be due to the user doing something that simply does not work. Your general Nextcloud knowledge might be helpful here - if this is the case, you can often swiftly close the issue with a comment about what went wrong.

Note: You might have to say no to some requests, for example when a problem has been solved in a new release but won't become available for the release the reporter is using; or when a solution has been chosen which the reporter is unhappy about. Be considerate. People feel surprisingly strong about Nextcloud, and you should take care to explain that we don't aim to ignore them; on the contrary. But sometimes, decisions which benefit the majority of users don't help an individual. The extensibility and open availability of the code of Nextcloud is here to relieve the pain of such decisions.

Determining if the report is complete

Now that you know that the bug report is unique, and that is not an external issue, you need to check all the needed information is there.

Check our bug reporting guidelines and make sure bug reports comply with it! The information asked in the issue template is needed for developers to solve issues.

Once you added a request for more information, add a #needinfo tag.

If there has been a request for more information on the report, either by you, a developer or somebody else, but the original reporter (or somebody else who might have the answer) has not responded for 1 month or longer, you can close the issue. Be polite and note that whoever can answer the question can re-open the issue!

Reproducing the issue

An important step of bug triaging is trying to reproduce the bugs, this means, using the information the reporters added to the bug report to force (recreate, reproduce, repeat) the bug in the application.

This is needed in order to differentiate random/race condition bugs of reproducible ones (which may be reproduced by developers too; and they can fix them).

If you can't reproduce an issue in a newer version of Nextcloud, it is most likely fixed and can be closed. Comment that you failed to reproduce the problem, and if the reporter can confirm (or doesn't respond for a long time), you can close the issue. Also, be sure to add what exactly you tested with - the Nextcloud Master or a branch (and if so, when), or did you use a release, and if so - what version?

Finalizing and tagging

Once you are done reproducing an issue, it is time to finish up and make clear to the developers what they can do:

- If it is a genuine bug (or you are pretty sure it is) add the 'bug' label.
- If it is a genuine feature request (or you are pretty sure it is) add the 'enhancement' label.
- If the issue is clearly related to something specific, set the specific feature label and @mention a maintainer

Now, the developers can pick the issue up. Note that while we wish we would always pick up and solve problems promptly, not all areas of Nextcloud get the same amount of attention and contribution, so this can occasionally take a long time.

Credit: this document is in debt to the extensive KDE guide to bug triaging.

Thank you for helping Nextcloud by reporting bugs. Before submitting an issue, please read Issue submission guidelines first.

- If the issue is with the Nextcloud server, report it to the Server repository
- If the issue is with the Nextcloud client, report it to the Client repository
- If the issue with with an Nextcloud app, report it to where that app is developed
- If the app is listed in our main GitHub organization report it to the correct sub repository

Help and Communication

Forums

Communicate via our forums.

IRC channels

Chat with us on IRC. All channels are on **irc.freenode.net**

- Setup: **#nextcloud**
- Development: **#nextcloud-dev**
- Design: **#nextcloud-design**
- Mobile: **#nextcloud-mobile**

Maintainers

- Contact a maintainer of a certain app or division