Nextcloud App Development Tutorials

Giuseppe Catillo

Nextcloud App Development Tutorials.. 1
- Setting up a development environment.. 3
  - How to stop the docker container?... 3
  - How to start the docker container a next time?.. 3
- Develop your first Hello World app.. 4
  - Overview of the skeleton files.. 4
  - Update skeleton files.. 5
- Develop a simple interface-only app.. 6
  - Write the frontend scripts... 6
    - Javascript dependency list in package.json.. 6
    - Javascript sources... 6
    - Webpack.. 6
    - Production frontend scripts.. 6
  - Define links routes and page... 7
    - Add the navigation item in the appinfo/info.xml file... 7
    - Set the app’s routes in the appinfo/routes.php file.. 7
    - Implement the PageController class.. 7
    - Write the template file... 7
    - Define the style.. 7
    - Compile the scripts... 7
- **Develop a dashboard widget with Vue.js.. 8**
  - Create a simple dashboard widget with Javascript... 8
    - Prepare the app skeleton... 8
    - Implement and register the dashboard widget.. 8
    - Frontend.. 8
    - Handle the network requests.. 9
    - Enable and test the app... 9
  - Create a dashboard widget using Vue.js components.. 10
    - Implement and register the dashboard widget.. 10
    - Frontend.. 10
    - Network requests.. 10
    - Enable and test the app... 10
- **Integrate an external provider using Smart Picker... 11**
  - Pull the latest development release of Nextcloud... 11
  - Install the required dependency apps... 11
  - Choose an external service that you can integrate... 11
  - Prepare the app skeleton.. 12
  - Implement the reference and search providers.. 12
  - Implement the admin settings.. 13
  - Install NPM dependencies.. 13
  - Write the admin settings scripts... 13
  - Write the reference widget script... 13
  - Compile the scripts.. 13
  - Handle the network requests... 14
  - Enable and test the app... 14

- **Develop a complete app with navigation bar and database........15**
    - Pull the latest development release of Nextcloud........15
    - Prepare the app skeleton........15
    - Set the lib/AppInfo/Application.php file........15
    - Set a basic route........15
    - Create lib/Controller/PageController.php and template/main.php........15
    - Install NPM dependencies........15
    - Create the front-end........15
    - Compile the scripts........17
    - Enable and test the app........17
    - Database migration........17
    - Database classes and mapper........17
        - Adjust the PageController........18
        - Add the configuration controller........18
    - Extend the OCS API........18
- **Adding automated tests to your app........19**
    - Prepare the app........19
    - Configure the tests........19
    - Implement the tests........19
    - Publish the app on Github and trigger the tests........20
- **Workflow Generico........21**

Setting up a development environment

1. Install the docker app
2. Install Nextcloud docker

Log-in credentials for Tutorial:

- username: admin
- password: admin

How to stop the docker container?

```
CTRL+C  +  docker compose down
```

(and press the enter key on your keyboard)

How to start the docker container a next time?

In the Terminal, cd to the folder nextcloud-docker-dev and run

```
$ sudo sysctl -w kernel.apparmor_restrict_unprivileged_userns=0
$ systemctl --user restart docker-desktop
$ docker compose up nextcloud proxy
```

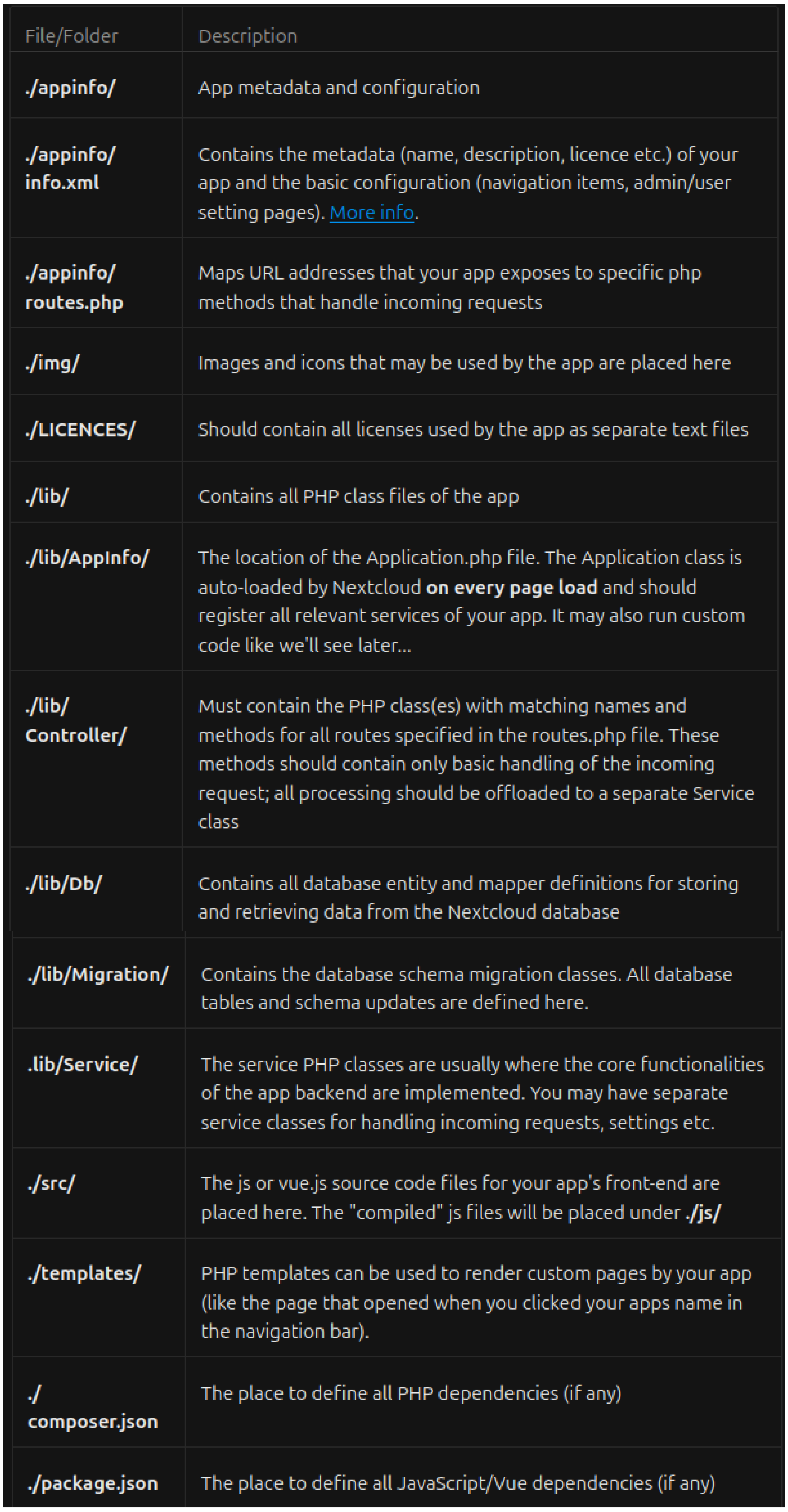
*temporary workaround for Ubuntu 24.04

- ## Develop your first Hello World app

Go to https://apps.nextcloud.com/developer/apps/generate

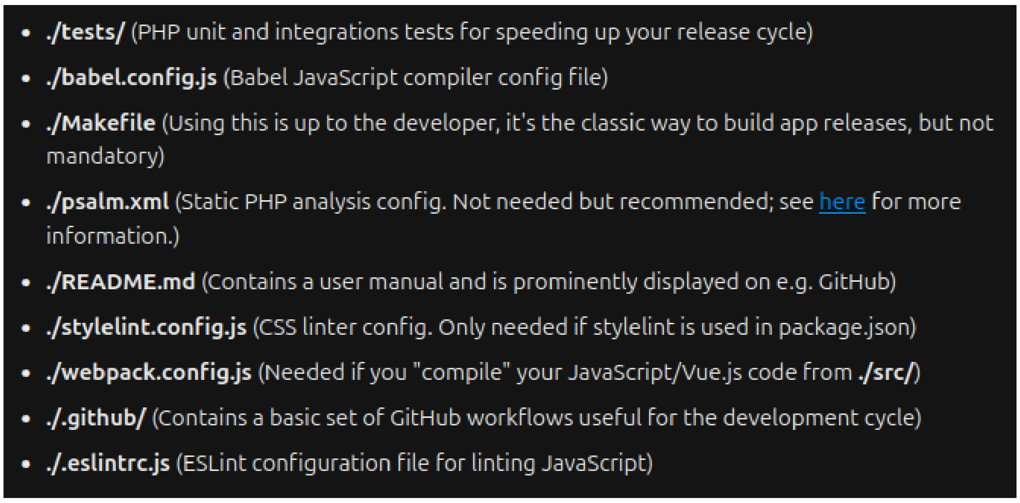
- Extract app.tar.gz
- Copy-paste the directory of your app in **workspace/server/apps-extra**
- Enable it by going to (Dashboard)->Apps->Your Apps->Find “Hello World” and click “Enable”

- ## Overview of the skeleton files



File/Folder	Description
./appinfo/	App metadata and configuration
./appinfo/info.xml	Contains the metadata (name, description, licence etc.) of your app and the basic configuration (navigation items, admin/user setting pages). More info.
./appinfo/routes.php	Maps URL addresses that your app exposes to specific php methods that handle incoming requests
./img/	Images and icons that may be used by the app are placed here
./LICENCES/	Should contain all licenses used by the app as separate text files
./lib/	Contains all PHP class files of the app
./lib/AppInfo/	The location of the Application.php file. The Application class is auto-loaded by Nextcloud **on every page load** and should register all relevant services of your app. It may also run custom code like we'll see later...
./lib/Controller/	Must contain the PHP class(es) with matching names and methods for all routes specified in the routes.php file. These methods should contain only basic handling of the incoming request; all processing should be offloaded to a separate Service class
./lib/Db/	Contains all database entity and mapper definitions for storing and retrieving data from the Nextcloud database
./lib/Migration/	Contains the database schema migration classes. All database tables and schema updates are defined here.
.lib/Service/	The service PHP classes are usually where the core functionalities of the app backend are implemented. You may have separate service classes for handling incoming requests, settings etc.
./src/	The js or vue.js source code files for your app's front-end are placed here. The "compiled" js files will be placed under **./js/**
./templates/	PHP templates can be used to render custom pages by your app (like the page that opened when you clicked your apps name in the navigation bar).
./composer.json	The place to define all PHP dependencies (if any)
./package.json	The place to define all JavaScript/Vue dependencies (if any)

Some files in the skeleton are only needed for setting up a **development pipeline**:



- **./tests/** (PHP unit and integrations tests for speeding up your release cycle)
- **./babel.config.js** (Babel JavaScript compiler config file)
- **./Makefile** (Using this is up to the developer, it's the classic way to build app releases, but not mandatory)
- **./psalm.xml** (Static PHP analysis config. Not needed but recommended; see here for more information.)
- **./README.md** (Contains a user manual and is prominently displayed on e.g. GitHub)
- **./stylelint.config.js** (CSS linter config. Only needed if stylelint is used in package.json)
- **./webpack.config.js** (Needed if you "compile" your JavaScript/Vue.js code from **./src/**)
- **./.github/** (Contains a basic set of GitHub workflows useful for the development cycle)
- **./.eslintrc.js** (ESLint configuration file for linting JavaScript)

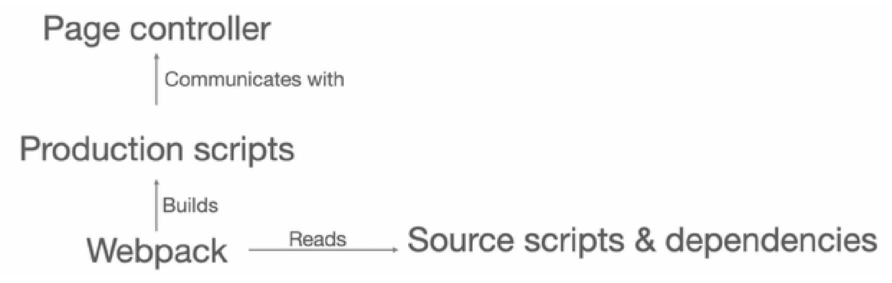
● Update skeleton files

- Open **./appinfo/info.xml** and adjust **max-version** to the Nextcloud version of your development (find it in Administration Settings->Overview)
- Define which **resources and routes** are needed or not for our app in **./appinfo/routes.php**
- Define ./lib/**Controller**/PageController.php and ./**templates**/main.php

● Develop a simple interface-only app

In this tutorial we will first install nvm, which is a Node Version Manager.
Through nvm we will install Node and NPM which are tools you need for **running and compiling javascript.**

● Write the frontend scripts



● Javascript dependency list in package.json

This JSON file contains all the dependencies that your source Javascript files need + some metadata:

- In a terminal, get **into the app folder** in your local Nextcloud setup and **install the dependency packages** with `npm install`

- Once the **package-lock.json** file exists, you only need to run `npm ci` (or `npm clean-install`) to install your dependencies again.

- Then **add @nextcloud/initial-state** as a dependency in both JSON files and install it with
  `npm i --save @nextcloud/initial-state`

● Javascript sources

These are the files you write to define what the **app's frontend** will do. Edit them inside the **src folder**.

● Webpack

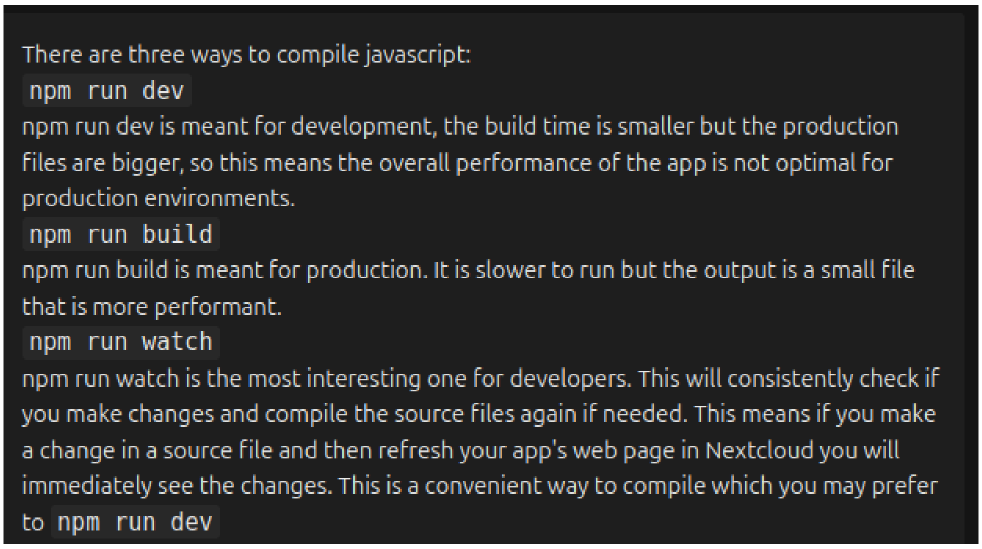
Our Javascript source file (and its dependencies) has to be compiled with Webpack to **produce the production files**.
Edit the file **webpack.config.js**

● Production frontend scripts

They are the **final scripts that are actually loaded in your app's frontend** (in the browser).

Run `npm run dev` to compile the javascript file:

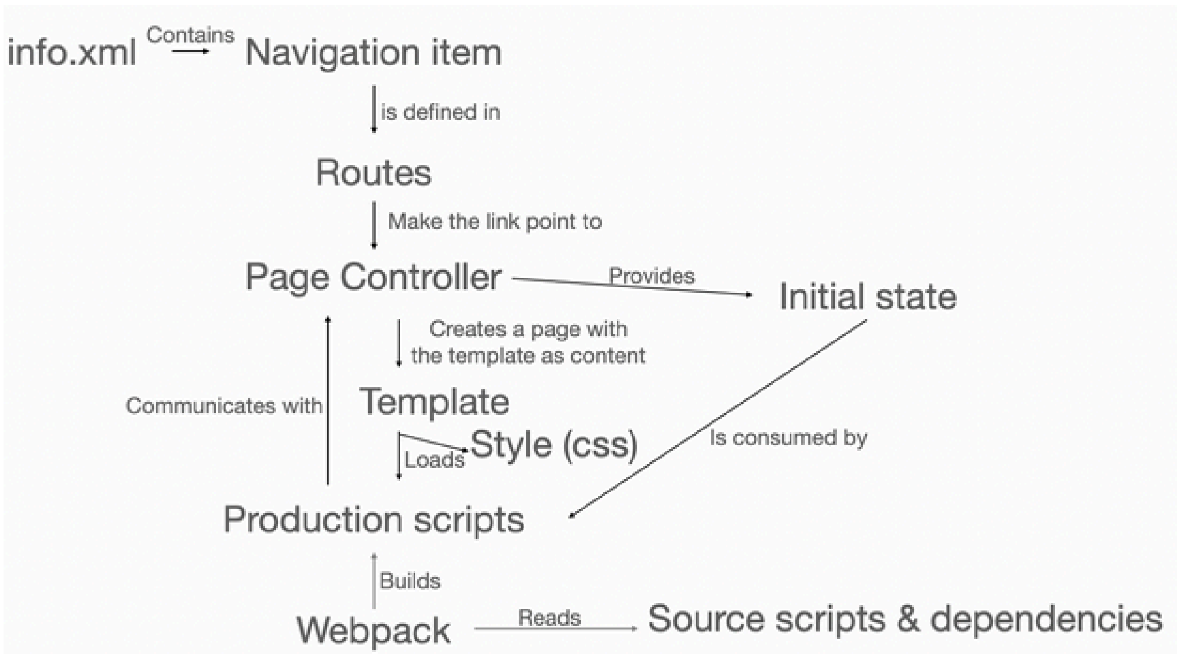
Se @nextcloud/dialogs non è presente digita `npm install \ @nextcloud/dialogs`



There are three ways to compile javascript:
`npm run dev`
npm run dev is meant for development, the build time is smaller but the production files are bigger, so this means the overall performance of the app is not optimal for production environments.
`npm run build`
npm run build is meant for production. It is slower to run but the output is a small file that is more performant.
`npm run watch`
npm run watch is the most interesting one for developers. This will consistently check if you make changes and compile the source files again if needed. This means if you make a change in a source file and then refresh your app's web page in Nextcloud you will immediately see the changes. This is a convenient way to compile which you may prefer to `npm run dev`

- ## Define links routes and page

In info.xml file we will define a navigation item with the **<route> tag**, whose value points to a c**ontroller method of our app** (or any other app), which will handle network requests and can answer with either some data (string, array, etc.) or a template**.**



- ### Add the navigation item in the appinfo/info.xml file

The <navigation> element defines the menu link.

- ### Set the app's routes in the appinfo/routes.php file

An app can define its own **API endpoints.** An endpoint has a route and a controller method to **handle the network requests**.

- ### Implement the PageController class

Edit **lib/AppInfo/Application.php** and **lib/Controller/PageController.php**.

If a route has been defined correctly in appinfo/routes.php, the target controller method handles the HTTP request and responds with some data or a template.

- ### Write the template file

Create a new **templates/**myMainTemplate.php file.

- ### Define the style

In your app's directory, create a **css/**main.css file

- ### Compile the scripts

As a precaution, let's **compile your source script again**, in case you made a change after the last time you compiled it, with `npm run build`

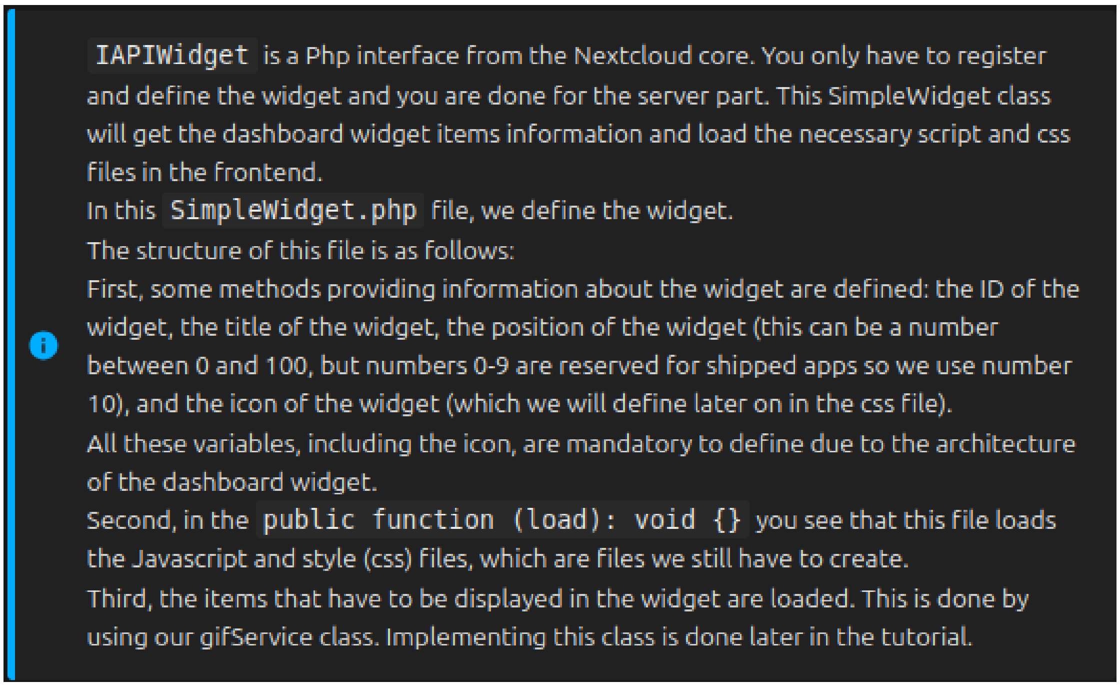
- **Develop a dashboard widget with Vue.js**
- **Create a simple dashboard widget with Javascript**
- **Prepare the app skeleton**

[…]
Create a **l10n** directory which will store the **translation files**.

Nextcloud provides mechanisms for internationalization (make an application translatable) and localization (add translations for specific languages), powered by Transifex.
To make this work with the automated translations you do need to configure this.

- **Implement and register the dashboard widget**

First, create the file **lib/Dashboard/SimpleWidget.php** and set its content:



`IAPIWidget` is a Php interface from the Nextcloud core. You only have to register and define the widget and you are done for the server part. This SimpleWidget class will get the dashboard widget items information and load the necessary script and css files in the frontend.
In this `SimpleWidget.php` file, we define the widget.
The structure of this file is as follows:
First, some methods providing information about the widget are defined: the ID of the widget, the title of the widget, the position of the widget (this can be a number between 0 and 100, but numbers 0-9 are reserved for shipped apps so we use number 10), and the icon of the widget (which we will define later on in the css file).
All these variables, including the icon, are mandatory to define due to the architecture of the dashboard widget.
Second, in the `public function (load): void {}` you see that this file loads the Javascript and style (css) files, which are files we still have to create.
Third, the items that have to be displayed in the widget are loaded. This is done by using our gifService class. Implementing this class is done later in the tutorial.

Then, register the dashboard widget in the **Application.php** file.

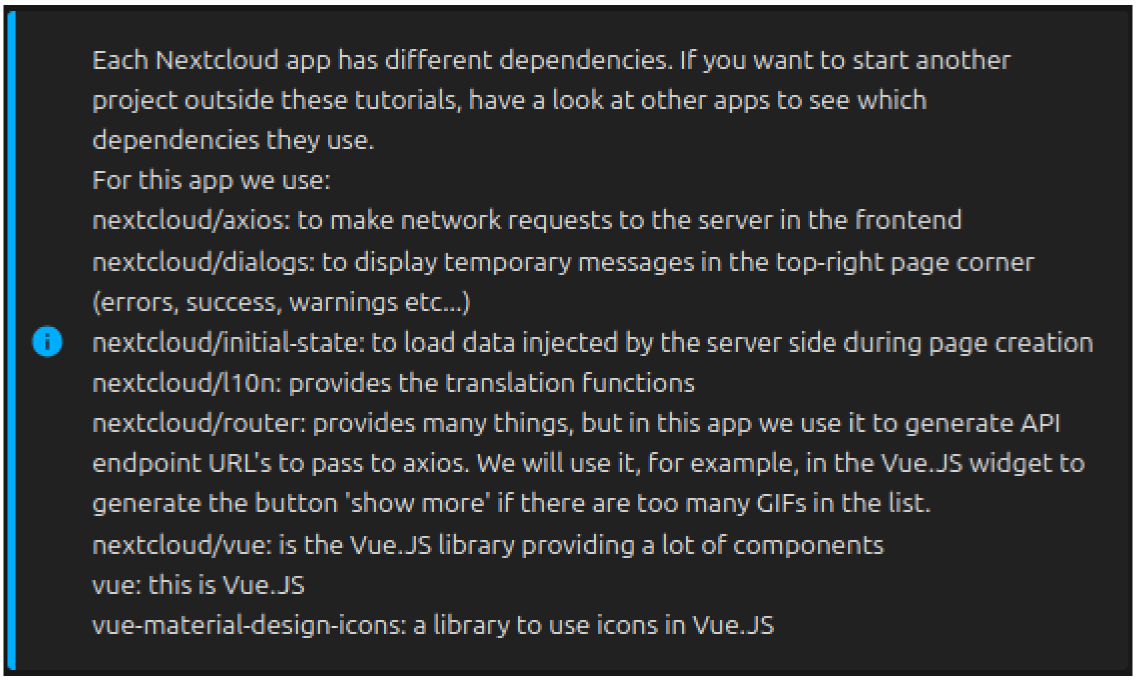
- **Frontend**

Make sure you are using **Node 16** by running `nvm use 16.16.0` (check with `node -v`)

Run `npm install` to **install the dependency packages** (to reinstall them you could just run `npm ci` without *npm install*)

Then execute the following command in order to **install the missing dependencies:**

- `npm i --save @nextcloud/axios@^2.3.0 @nextcloud/dialogs@^3.2.0 @nextcloud/initial-state@^2.0.0 @nextcloud/l10n@^2.0.1 @nextcloud/router@^2.0.1 @nextcloud/vue@^7.5.0 vue@^2.7.14 vue-material-design-icons@^5.2.0`
- `npm i --save-dev eslint-webpack-plugin@^4.0.0 stylelint-webpack-plugin@^4.0.0`



Each Nextcloud app has different dependencies. If you want to start another project outside these tutorials, have a look at other apps to see which dependencies they use.
For this app we use:
nextcloud/axios: to make network requests to the server in the frontend
nextcloud/dialogs: to display temporary messages in the top-right page corner (errors, success, warnings etc...)
nextcloud/initial-state: to load data injected by the server side during page creation
nextcloud/l10n: provides the translation functions
nextcloud/router: provides many things, but in this app we use it to generate API endpoint URL's to pass to axios. We will use it, for example, in the Vue.JS widget to generate the button 'show more' if there are too many GIFs in the list.
nextcloud/vue: is the Vue.JS library providing a lot of components
vue: this is Vue.JS
vue-material-design-icons: a library to use icons in Vue.JS

- Create the **Javascript source file** src/dashboardSimple.js of the simple dashboard widget that will actually be loaded by the SimpleWidget class and executed in the dashboard page!

- **Configure Webpack**: set the content of the webpack.config.js file

- Run `npm run dev` to **compile** the Javascript source files:
  The error about '**Newline required at the end of file but not found**' means you have to add an empty line at the end of the dashboardSimple.js file.

- ## Handle the network requests

Make sure that Nextcloud knows which controller method to execute when receiving network requests by setting the **app's routes,** in the appinfo/routes.php file.

Then **implement the controller** in the directory lib/Controller, which will handle network requests and respond with data or a template.

To make the file more readable we choose to split the implementation of retrieving those GIF files into a separate class **GifService**, specified in **lib/Service/**GifService.php:
    the service does the action and the controller is only receiving the request and building the response.

- ## Enable and test the app

- **Enable** the Cat Gifs Dashboard app in app settings.

- (Create a gifs directory in the files app and add your favorite cat GIFs in this director)

- Go to the Dashboard app and click the **Customize** button: **Enable the Simple Widget**

● Create a dashboard widget using Vue.js components

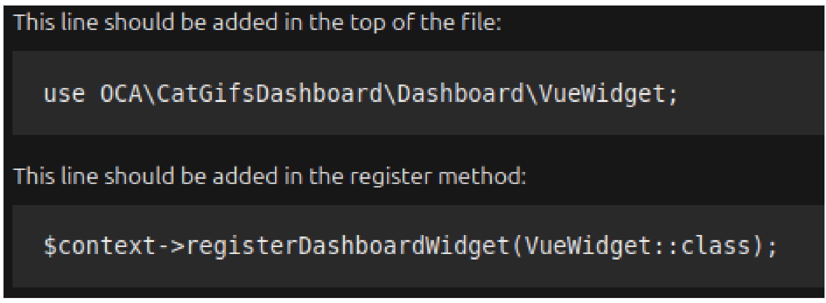
On top of HTML we can load scripts that read and write HTML content.
Frontend scripts (in Javascript) are programs that run in the browser and it is hard to make complex things in an elegant way.

With Vue.js you can create **components** that you can take and put everywhere you want, even in other pages or event components, each one with a template (the HTML), a script section and a style section.

● Implement and register the dashboard widget

Create the file **lib/Dashboard/VueWidget.php** and set its content.

Then, **register the dashboard widget** in the Application.php file with these lines:



This line should be added in the top of the file:

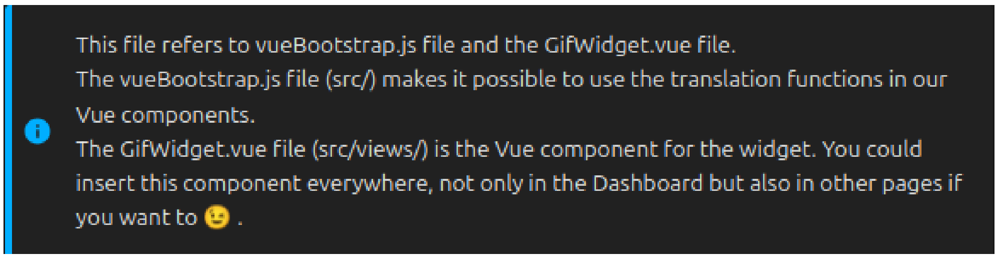
```
use OCA\CatGifsDashboard\Dashboard\VueWidget;
```

This line should be added in the register method:

```
$context->registerDashboardWidget(VueWidget::class);
```

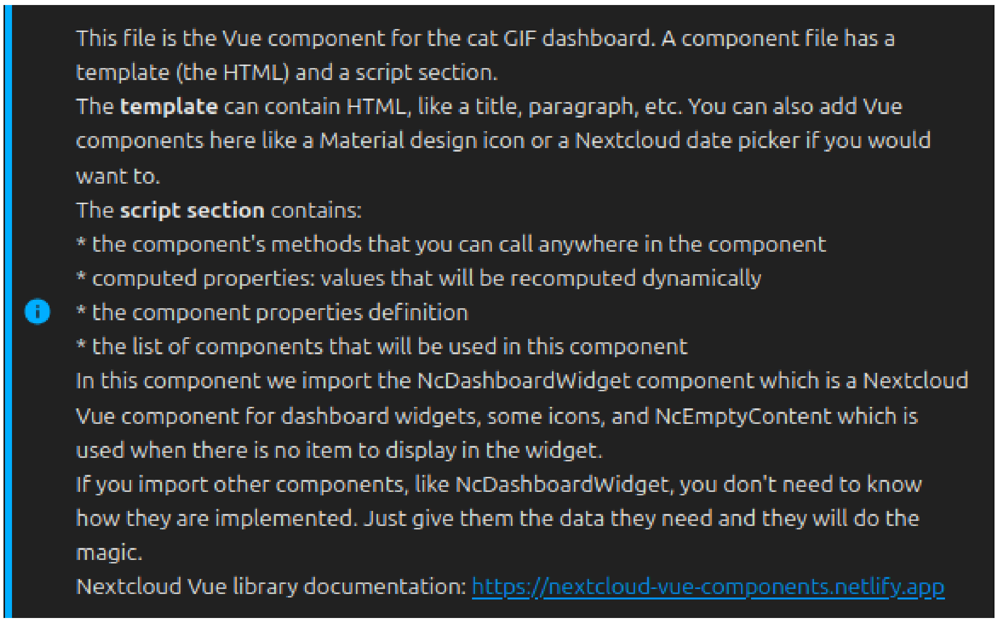
● Frontend

- Write the **Javascript source file of the Vue dashboard widget** in the src folder



This file refers to vueBootstrap.js file and the GifWidget.vue file.
The vueBootstrap.js file (src/) makes it possible to use the translation functions in our Vue components.
The GifWidget.vue file (src/views/) is the Vue component for the widget. You could insert this component everywhere, not only in the Dashboard but also in other pages if you want to 😉 .

- (in this example) configure the **Vue component**:



This file is the Vue component for the cat GIF dashboard. A component file has a template (the HTML) and a script section.
The **template** can contain HTML, like a title, paragraph, etc. You can also add Vue components here like a Material design icon or a Nextcloud date picker if you would want to.
The **script section** contains:
* the component's methods that you can call anywhere in the component
* computed properties: values that will be recomputed dynamically
* the component properties definition
* the list of components that will be used in this component
In this component we import the NcDashboardWidget component which is a Nextcloud Vue component for dashboard widgets, some icons, and NcEmptyContent which is used when there is no item to display in the widget.
If you import other components, like NcDashboardWidget, you don't need to know how they are implemented. Just give them the data they need and they will do the magic.
Nextcloud Vue library documentation: https://nextcloud-vue-components.netlify.app

- Configure **Webpack**
- **Compile the source files** with `npm run dev`

● Network requests

● Enable and test the app

Refresh the Nextcloud dashboard page → Click the **Customize** button → **Enable** the Vue Widget.

- ## Integrate an external provider using Smart Picker

- ### Pull the latest development release of Nextcloud

First you need to develop on the **latest development environment**, so go into the nextcloud/workspace/server directory run:

```
git pull origin master
git submodule update
```

Reload your local Nextcloud without the cache and click the ‘update’ button.
If you receive an error for a specific app, solve it by running `git pull` in each of the app directories.

- ### Install the required dependency apps

The Smart Picker (introduced in Nextcloud 26) is one of the best components to use to quickly **create integrations that also interact with many different Nextcloud apps**.

In this tutorial we will go through the steps of creating an app which extends the Smart Picker with a provider that gets data from an external service.
The Smart Picker can be used in Talk, Text, Collectives, Deck, Notes, and Mail. You need at least one of these apps to be able to test your app. We recommend you to use **Text** for this.

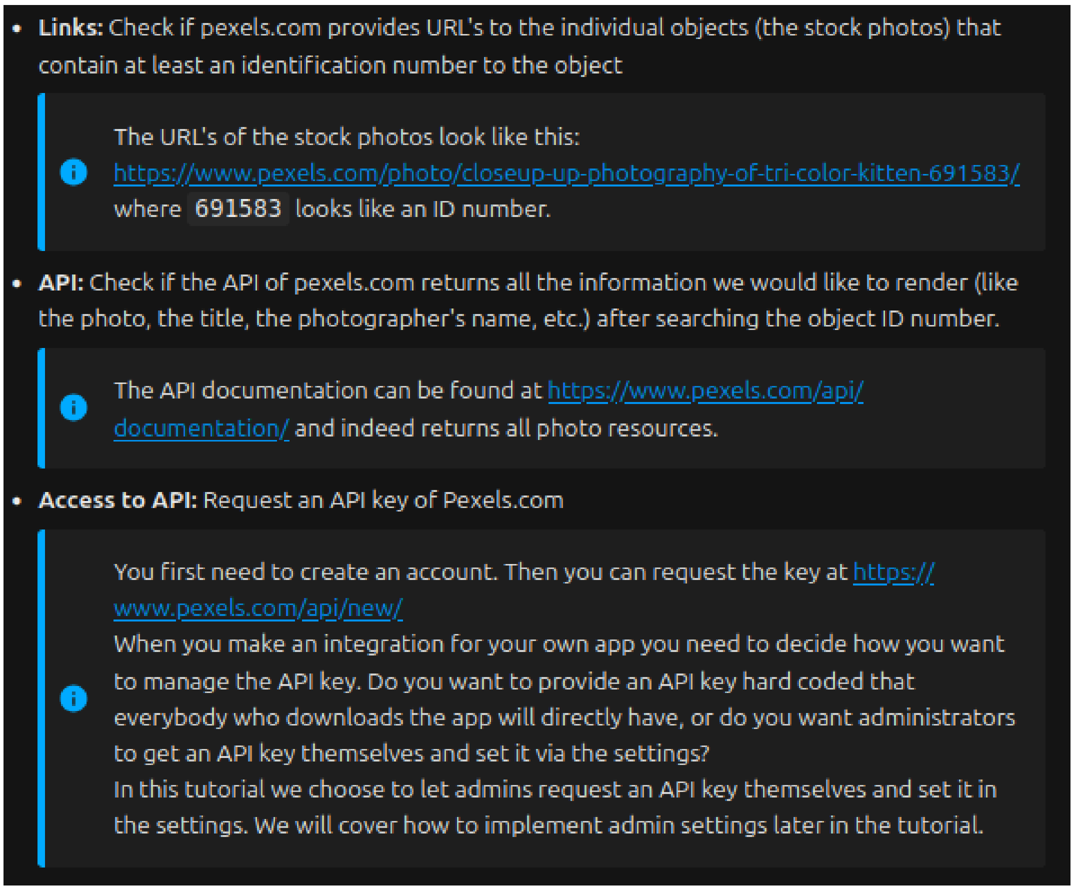
Run in the **workspace/server/apps-extra** directory:

```
git clone https://github.com/nextcloud/text
```

Reload nextcloud.local without the cache and in the apps settings **enable Text**.

- ### Choose an external service that you can integrate

We will integrate the stock photo site Pexels.com, but these steps apply to any external provider.
When you have an idea to integrate an external service, there are **three things you need to check:**



- **Links:** Check if pexels.com provides URL's to the individual objects (the stock photos) that contain at least an identification number to the object

  > The URL's of the stock photos look like this:
  > https://www.pexels.com/photo/closeup-up-photography-of-tri-color-kitten-691583/
  > where `691583` looks like an ID number.

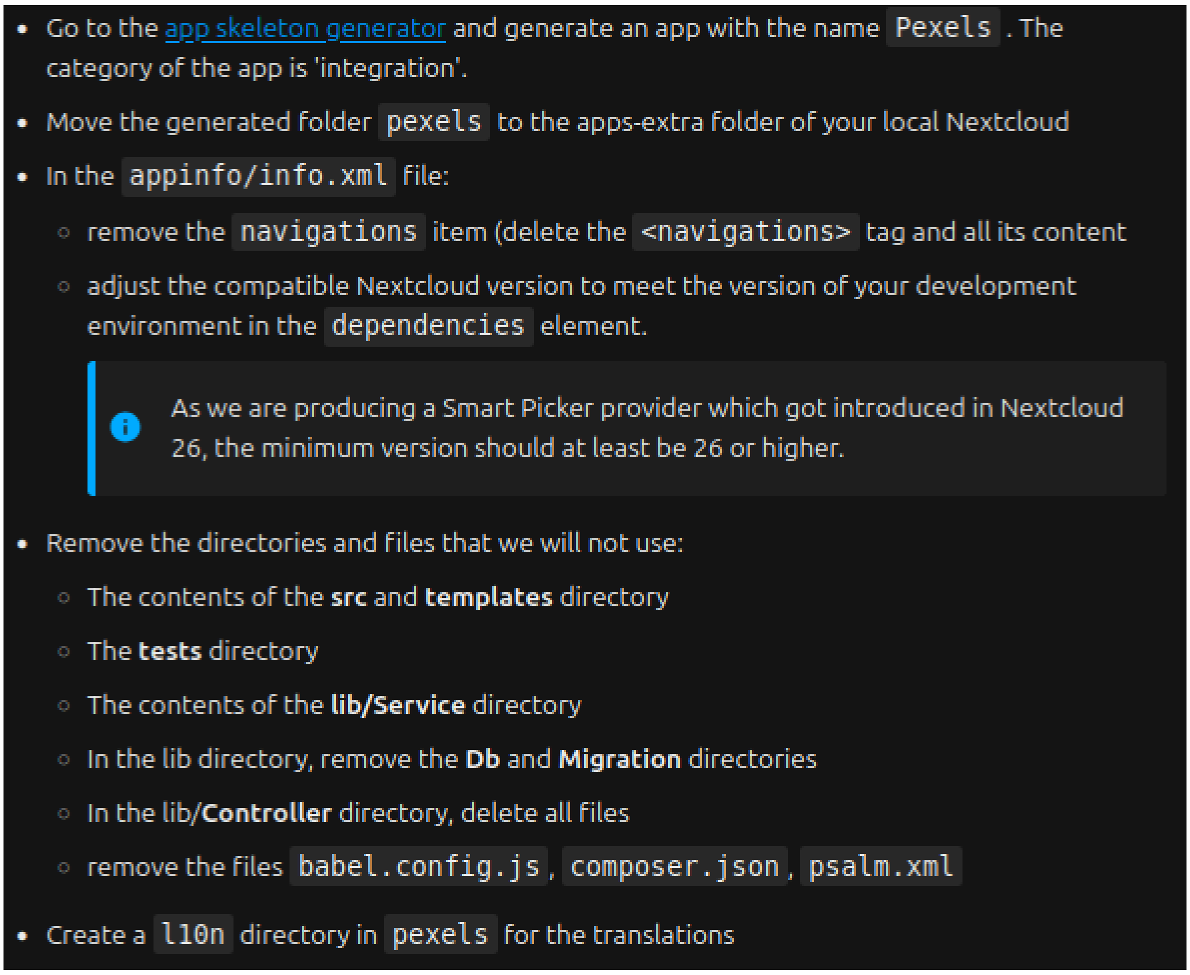
- **API:** Check if the API of pexels.com returns all the information we would like to render (like the photo, the title, the photographer's name, etc.) after searching the object ID number.

  > The API documentation can be found at https://www.pexels.com/api/documentation/ and indeed returns all photo resources.

- **Access to API:** Request an API key of Pexels.com

  > You first need to create an account. Then you can request the key at https://www.pexels.com/api/new/
  > When you make an integration for your own app you need to decide how you want to manage the API key. Do you want to provide an API key hard coded that everybody who downloads the app will directly have, or do you want administrators to get an API key themselves and set it via the settings?
  > In this tutorial we choose to let admins request an API key themselves and set it in the settings. We will cover how to implement admin settings later in the tutorial.

- ## Prepare the app skeleton



- Go to the app skeleton generator and generate an app with the name `Pexels`. The category of the app is 'integration'.
- Move the generated folder `pexels` to the apps-extra folder of your local Nextcloud
- In the `appinfo/info.xml` file:
  - remove the `navigations` item (delete the `<navigations>` tag and all its content
  - adjust the compatible Nextcloud version to meet the version of your development environment in the `dependencies` element.

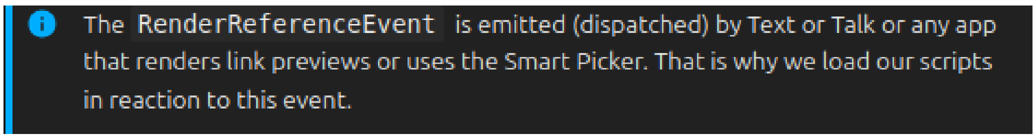
    As we are producing a Smart Picker provider which got introduced in Nextcloud 26, the minimum version should at least be 26 or higher.
- Remove the directories and files that we will not use:
  - The contents of the **src** and **templates** directory
  - The **tests** directory
  - The contents of the **lib/Service** directory
  - In the lib directory, remove the **Db** and **Migration** directories
  - In the lib/**Controller** directory, delete all files
  - remove the files `babel.config.js`, `composer.json`, `psalm.xml`
- Create a `l10n` directory in `pexels` for the translations

- ## Implement the reference and search providers

Similar to a dashboard widget, a reference provider and search provider is **first implemented as a class and then registered** in the lib/AppInfo/Application.php file.

- Implement a **reference provider** which will resolve the links.
  Create the **lib/Reference directory** and the lib/Reference/PhotoReferenceProvider.php file

- Implement an **event listener** which will react to the RenderReferenceEvent to load the scripts that will register the reference widget component in the frontend.

  The `RenderReferenceEvent` is emitted (dispatched) by Text or Talk or any app that renders link previews or uses the Smart Picker. That is why we load our scripts in reaction to this event.

  Create the **lib/Listener directory** and the lib/Listener/PexelsReferenceListener.php file.

- Implement the **search provider** in order to enable you to extend the Nextcloud unified search in the top-right of Nextcloud to also search in Pexels.
  Create the **lib/Search directory** and the lib/Search/PexelsSearchPhotosProvider.php file.

  There is no need to implement a user interface for a search provider.
  The server generic search provider menu will render our search results.

- ## Implement the admin settings

Create an **administration settings section** (new navigation menu item) where the administrator will be able to set an API key for the Pexels integration:

- Create the **lib/Settings directory** and the lib/Settings/AdminSection.php file.

- Implement the class that will get our **settings values** (the API key) and provide them to the frontend inside the lib/Settings/Admin.php file.

- Implement the **adminSettings template** inside the file templates/adminSettings.php

- Then, **register these administration settings** in **appinfo/info.xml** by adding the <settings> tag after the <dependencies> tag.

- ## Install NPM dependencies

[Make sure you are using Node 16 and all its dependencies]

- ## Write the admin settings scripts

The search provider implementation does not need any script.
The implementation of the **admin settings and then the reference widget (provider)** do need scripts.

- Create the **src/adminSettings.js** file

- Implement the Vue.js **component** which will be used in the admin settings by creating the directory **src/components** and the file src/components/AdminSettings.vue

- Implement the **icon component** that is needed for the admin settings by creating the directory src/components/icons and the file src/components/icons/PexelsIcon.vue

- ## Write the reference widget script

- Implement the script to **register the reference widget** inside the file src/reference.js

- Implement the PhotoReferenceWidget **component** that is needed for rendering the Pexels links, inside the src/views/PhotoReferenceWidget.vue file

- ## Compile the scripts

- Edit the **webpack.config.js**.

- **Compile the Javascript source files** with `npm run dev`
  (if you encounter any problems/warnings fix them and run `npm run dev`)

Handle the network requests

Make sure that Nextcloud knows which **controller method** to execute when receiving network requests by setting the app's routes in the **appinfo/routes.php**.

- Implement the **Controllers classes** inside the lib/Controller/ directory.

- Implement the **Service class** inside the lib/Service/ directory, which communicates with the API and could be used by any controller or any other service in our app.

Enable and test the app

- **Enable** the app in app settings

- **Add the API key** in your administration settings in the menu item “Connected accounts”

- **Open Text** and type / .
  This will open the Smart Picker. Select the Pexels Smart Picker and enjoy inserting stock photos.

Develop a complete app with navigation bar and database

Pull the latest development release of Nextcloud

Prepare the app skeleton

- **App skeleton generator**
- Move the generated folder to the **apps-extra folder** of your local Nextcloud
- Adjust the **compatible Nextcloud version** in the dependencies element of **appinfo/info.xml**
- Create a **l10n** directory for the translations
- [Adjust icon by replacing the files in img directory]

Set the lib/AppInfo/Application.php file

The Application.php file contains the dynamic declaration of an app.
It defines what the app does in Nextcloud in general, on the server side.

Set a basic route

Each app can define its own API endpoints.
An endpoint has a route and a controller method to handle the network requests.

Create lib/Controller/PageController.php and template/main.php

Install NPM dependencies

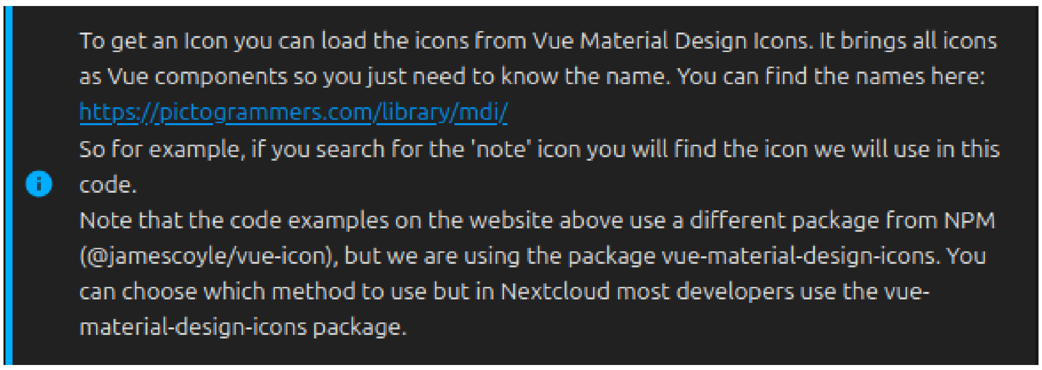
[Make sure you are using Node 16 and all its dependencies]

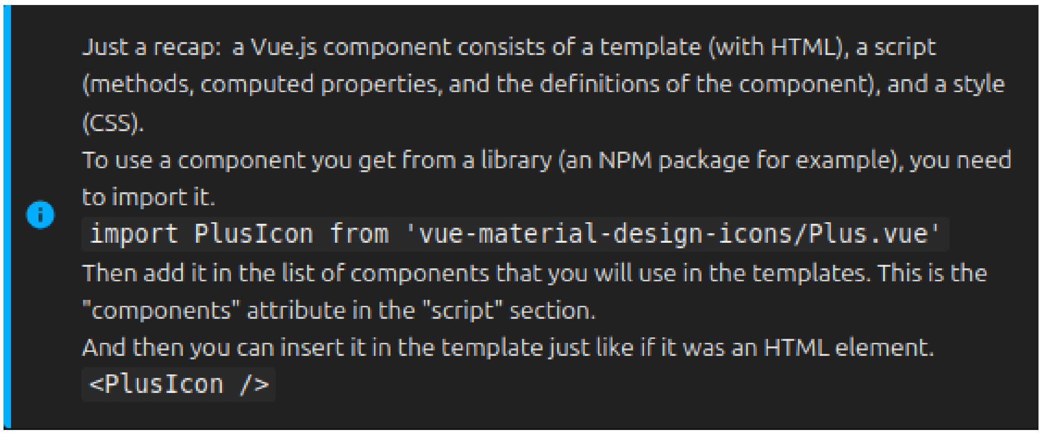
It is possible to **adjust the .eslintrc.js file**, which checks the code before it is being compiled through some rules.

Create the front-end

The front-end of this app is created in Vue, so we need to **'mount' our top Vue component** (App.vue) in an HTML element inside **src/main.js** and actually create the file **src/views/App.vue**.

Then inside the **src/components** directory we implement the Vue.js components, along with thei icons in the src/components/icons directory.

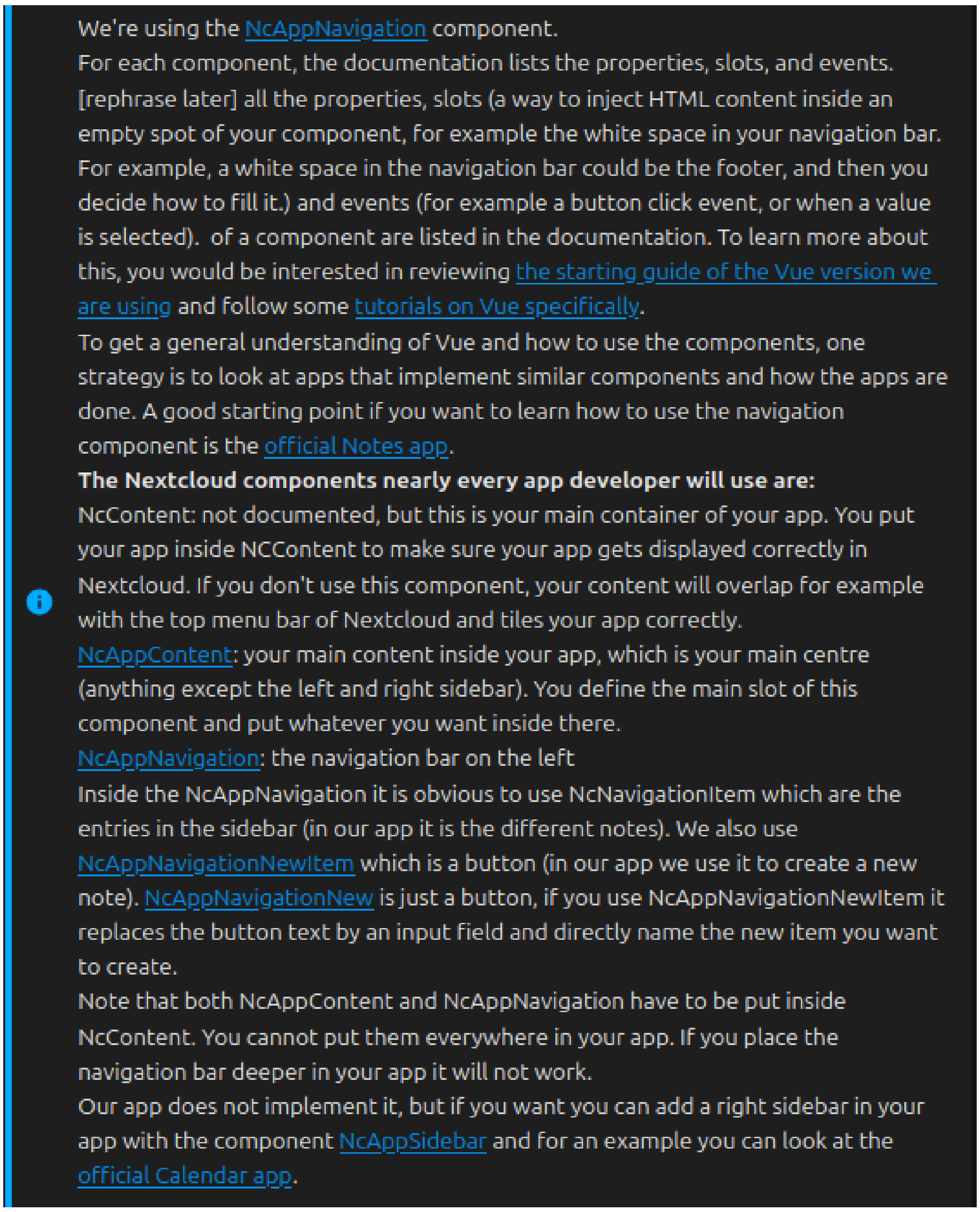

To get an Icon you can load the icons from Vue Material Design Icons. It brings all icons as Vue components so you just need to know the name. You can find the names here: https://pictogrammers.com/library/mdi/
So for example, if you search for the 'note' icon you will find the icon we will use in this code.
Note that the code examples on the website above use a different package from NPM (@jamescoyle/vue-icon), but we are using the package vue-material-design-icons. You can choose which method to use but in Nextcloud most developers use the vue-material-design-icons package.


Just a recap: a Vue.js component consists of a template (with HTML), a script (methods, computed properties, and the definitions of the component), and a style (CSS).
To use a component you get from a library (an NPM package for example), you need to import it.

```
import PlusIcon from 'vue-material-design-icons/Plus.vue'
```

Then add it in the list of components that you will use in the templates. This is the "components" attribute in the "script" section.
And then you can insert it in the template just like if it was an HTML element.

```
<PlusIcon />
```


We're using the NcAppNavigation component.
For each component, the documentation lists the properties, slots, and events.
[rephrase later] all the properties, slots (a way to inject HTML content inside an empty spot of your component, for example the white space in your navigation bar. For example, a white space in the navigation bar could be the footer, and then you decide how to fill it.) and events (for example a button click event, or when a value is selected). of a component are listed in the documentation. To learn more about this, you would be interested in reviewing the starting guide of the Vue version we are using and follow some tutorials on Vue specifically.
To get a general understanding of Vue and how to use the components, one strategy is to look at apps that implement similar components and how the apps are done. A good starting point if you want to learn how to use the navigation component is the official Notes app.
The Nextcloud components nearly every app developer will use are:
NcContent: not documented, but this is your main container of your app. You put your app inside NCContent to make sure your app gets displayed correctly in Nextcloud. If you don't use this component, your content will overlap for example with the top menu bar of Nextcloud and tiles your app correctly.
NcAppContent: your main content inside your app, which is your main centre (anything except the left and right sidebar). You define the main slot of this component and put whatever you want inside there.
NcAppNavigation: the navigation bar on the left
Inside the NcAppNavigation it is obvious to use NcNavigationItem which are the entries in the sidebar (in our app it is the different notes). We also use NcAppNavigationNewItem which is a button (in our app we use it to create a new note). NcAppNavigationNew is just a button, if you use NcAppNavigationNewItem it replaces the button text by an input field and directly name the new item you want to create.
Note that both NcAppContent and NcAppNavigation have to be put inside NcContent. You cannot put them everywhere in your app. If you place the navigation bar deeper in your app it will not work.
Our app does not implement it, but if you want you can add a right sidebar in your app with the component NcAppSidebar and for an example you can look at the official Calendar app.

If a component uses a script, define it inside src/ directory.

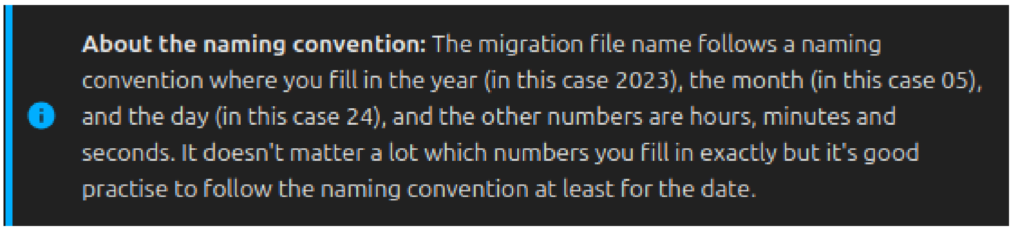
Compile the scripts

Enable and test the app

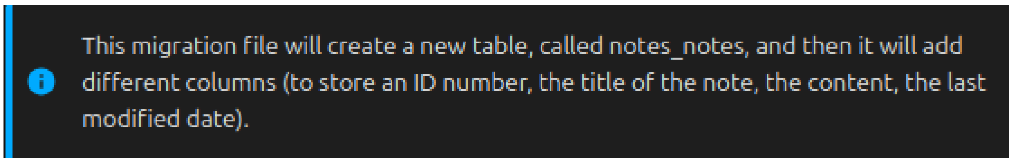
(Right now, the app will not yet do anything, it's just an interface-only app displaying the navigation bar which will be empty)

Database migration

Create the **migration file** inside the **lib/Migration** directory following this naming convention:

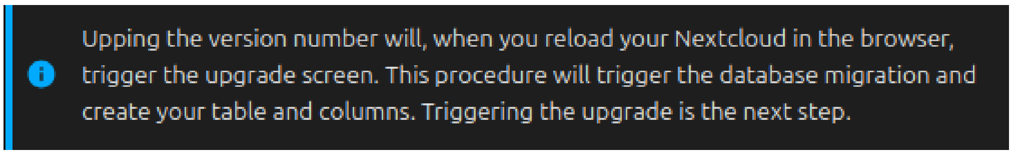


About the naming convention: The migration file name follows a naming convention where you fill in the year (in this case 2023), the month (in this case 05), and the day (in this case 24), and the other numbers are hours, minutes and seconds. It doesn't matter a lot which numbers you fill in exactly but it's good practise to follow the naming convention at least for the date.



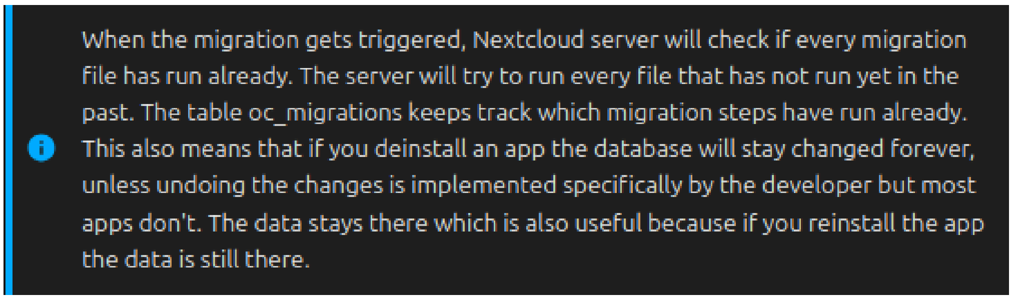
This migration file will create a new table, called notes_notes, and then it will add different columns (to store an ID number, the title of the note, the content, the last modified date).

In the **appinfo/info.xml** file, **up the version number** (<version> tag).



Upping the version number will, when you reload your Nextcloud in the browser, trigger the upgrade screen. This procedure will trigger the database migration and create your table and columns. Triggering the upgrade is the next step.

Then hard refresh the Nextcloud instance.



When the migration gets triggered, Nextcloud server will check if every migration file has run already. The server will try to run every file that has not run yet in the past. The table oc_migrations keeps track which migration steps have run already. This also means that if you deinstall an app the database will stay changed forever, unless undoing the changes is implemented specifically by the developer but most apps don't. The data stays there which is also useful because if you reinstall the app the data is still there.

Database classes and mapper

One way to interact with the database would be to directly write SQL queries in php, but this is not the recommended way.
The recommended way in Nextcloud is to interact with the database through an abstraction layer.

To create this abstraction layer, inside the **lib/Db** directory….

- First create a **class for each table**
- Then define a **mapper** (*class_name*Mapper.php) for each class, which specifies the operations that you can do in the database table.
  This is the bridge between the database and the programming language and it can be used anywhere in the app (controllers, background ecc…)

- ### Adjust the PageController

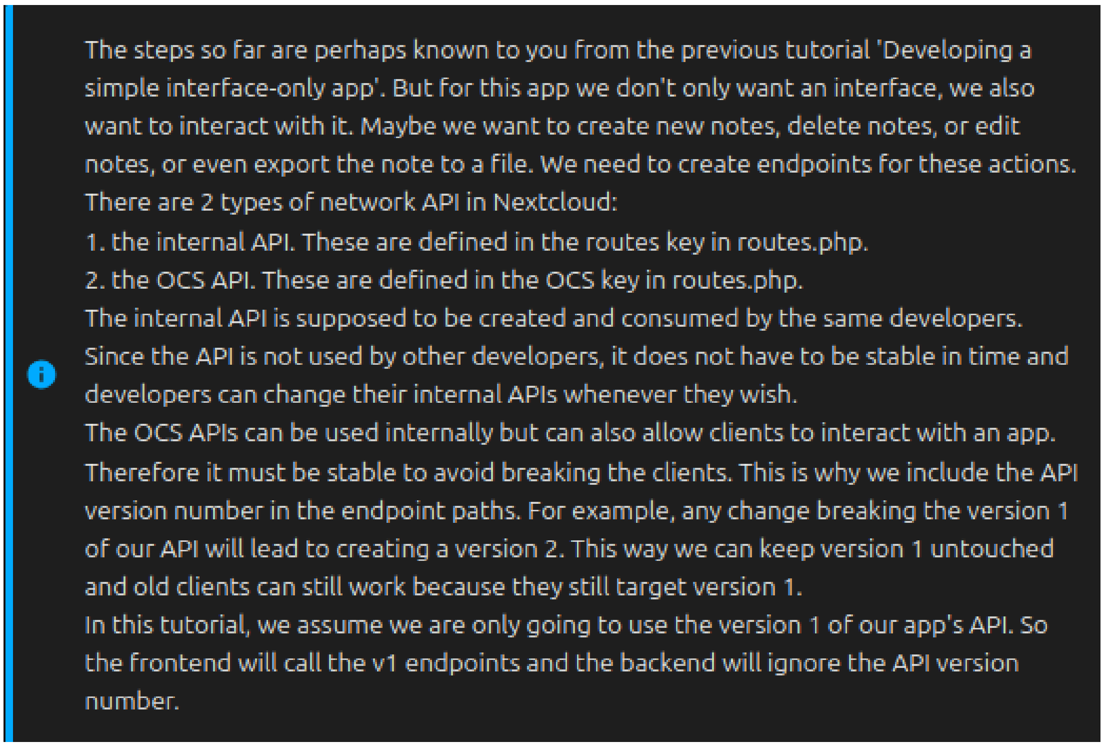
We need something that actually uses the mapper so inside **lib/Controller/PageController.php**:

- Add the declaration to use the mapper (*use OCA\Notes\Db\NoteMapper*);
- Declare the class (*private NoteMapper $noteMapper,*);
- Extend the controller to use it *($notes = $this->noteMapper->getNotesOfUser($this->userId));*

- ### Add the configuration controller

Define a route (appinfo/routes.php) and a class (lib/Controller/…) for the configuration controller.

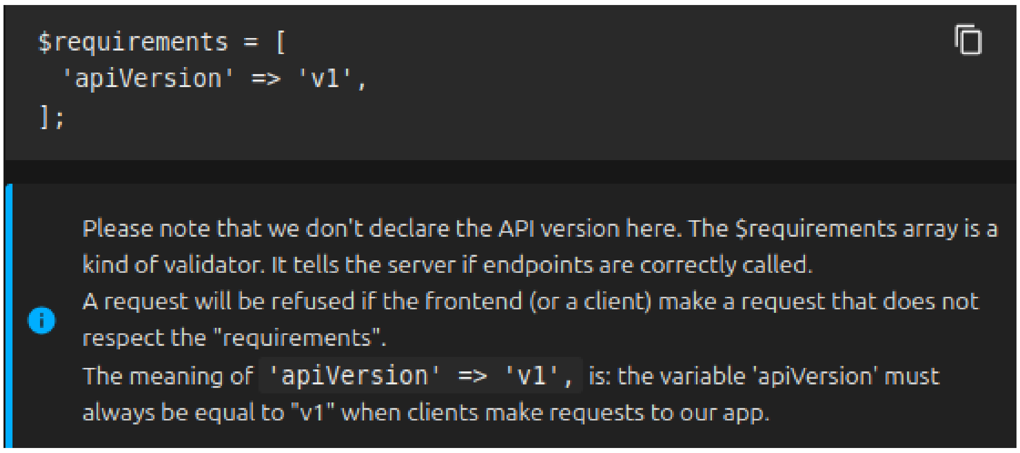
- ## Extend the OCS API



The steps so far are perhaps known to you from the previous tutorial 'Developing a simple interface-only app'. But for this app we don't only want an interface, we also want to interact with it. Maybe we want to create new notes, delete notes, or edit notes, or even export the note to a file. We need to create endpoints for these actions.
There are 2 types of network API in Nextcloud:
1. the internal API. These are defined in the routes key in routes.php.
2. the OCS API. These are defined in the OCS key in routes.php.
The internal API is supposed to be created and consumed by the same developers. Since the API is not used by other developers, it does not have to be stable in time and developers can change their internal APIs whenever they wish.
The OCS APIs can be used internally but can also allow clients to interact with an app. Therefore it must be stable to avoid breaking the clients. This is why we include the API version number in the endpoint paths. For example, any change breaking the version 1 of our API will lead to creating a version 2. This way we can keep version 1 untouched and old clients can still work because they still target version 1.
In this tutorial, we assume we are only going to use the version 1 of our app's API. So the frontend will call the v1 endpoints and the backend will ignore the API version number.

In the appinfo/routes.php file…

- First **restrict the API** version when the endpoints are requested by adding the following content



```
$requirements = [
  'apiVersion' => 'v1',
];
```

Please note that we don't declare the API version here. The $requirements array is a kind of validator. It tells the server if endpoints are correctly called.
A request will be refused if the frontend (or a client) make a request that does not respect the "requirements".
The meaning of `'apiVersion' => 'v1',` is: the variable 'apiVersion' must always be equal to "v1" when clients make requests to our app.

- Then **declare the API endpoints** by adding the *'ocs'* content in the array that is returned.

Define any “global variable” in lib/AppInfo/Application.php and reference it with *Application::<name>*

Create the Controller inside lib/Controller and the Service inside lib/Service, in case you want to improve the readability of the code.

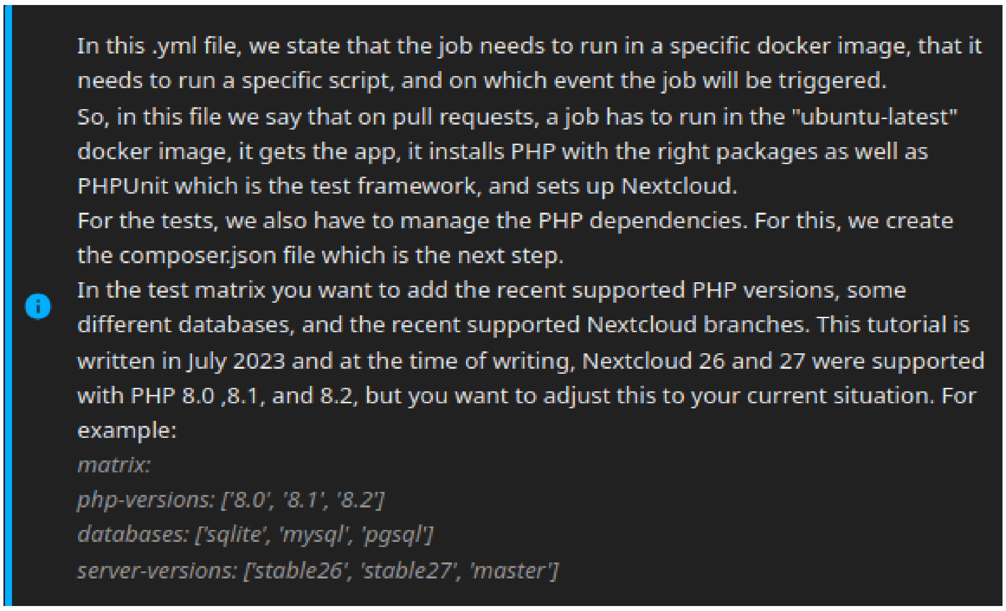
Adding automated tests to your app

Prepare the app

(We will add tests for the previous notebook app)
Go to the **.github** directory and delete all the files in it and in the **workflows** directory:
in the end just keep the workflows directory empty

Configure the tests

On GitHub you can run **actions on some events** by specifying them in .github/workflows/**phpunit.yml**



In this .yml file, we state that the job needs to run in a specific docker image, that it needs to run a specific script, and on which event the job will be triggered.
So, in this file we say that on pull requests, a job has to run in the "ubuntu-latest" docker image, it gets the app, it installs PHP with the right packages as well as PHPUnit which is the test framework, and sets up Nextcloud.
For the tests, we also have to manage the PHP dependencies. For this, we create the composer.json file which is the next step.
In the test matrix you want to add the recent supported PHP versions, some different databases, and the recent supported Nextcloud branches. This tutorial is written in July 2023 and at the time of writing, Nextcloud 26 and 27 were supported with PHP 8.0 ,8.1, and 8.2, but you want to adjust this to your current situation. For example:

```
matrix:
php-versions: ['8.0', '8.1', '8.2']
databases: ['sqlite', 'mysql', 'pgsql']
server-versions: ['stable26', 'stable27', 'master']
```

** add stable30 to server-versions and consequently 8.3 to php-versions.*

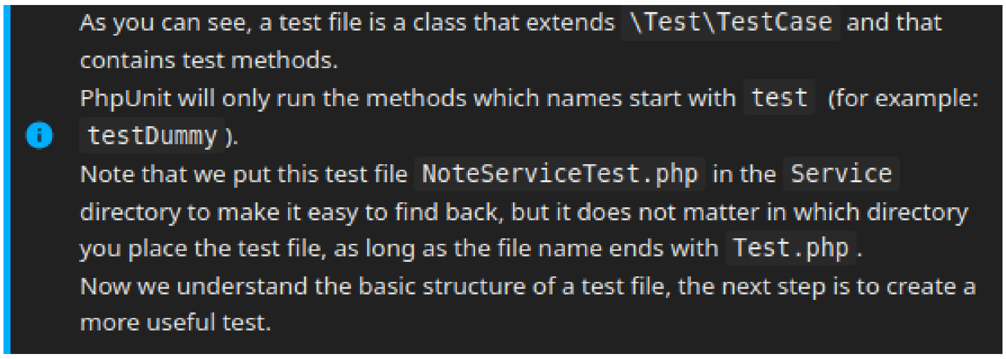
Edit the **composer.json** file

The GitHub action will run the tests that are configured in the **tests/phpunit.xml** file, which specifies the PHPUnit configuration.
Also create the **tests/bootstrap.php** file, needed to run the tests in a proper environment.

Implement the tests

Create the directory **tests/unit**:

- Create the file tests/unit/**Service/NoteServiceTest.php**



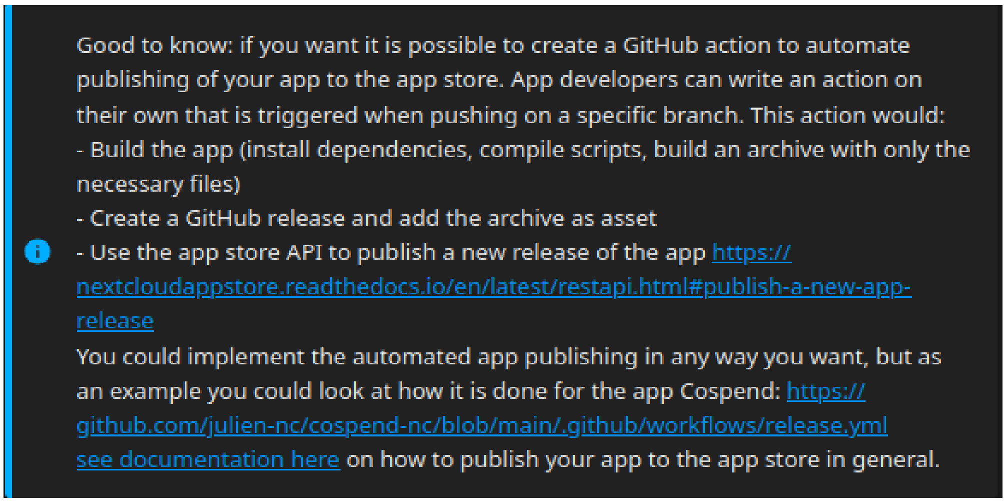
As you can see, a test file is a class that extends `\Test\TestCase` and that contains test methods.
PhpUnit will only run the methods which names start with `test` (for example: `testDummy`).
Note that we put this test file `NoteServiceTest.php` in the `Service` directory to make it easy to find back, but it does not matter in which directory you place the test file, as long as the file name ends with `Test.php` .
Now we understand the basic structure of a test file, the next step is to create a more useful test.

- Create the file tests/unit/**Mapper/NoteMapperTest.php**:
  this file contains all the test cases

Publish the app on Github and trigger the tests

Publish your app to GitHub and then create a small **pull request to trigger the tests.**

In GitHub, you can access the test results in the **Actions tab** by clicking on any "workflow run".
You can then see if all the "jobs" ran successfully and you can find the details of the test logged to the phpunit.xml file.



Good to know: if you want it is possible to create a GitHub action to automate publishing of your app to the app store. App developers can write an action on their own that is triggered when pushing on a specific branch. This action would:
- Build the app (install dependencies, compile scripts, build an archive with only the necessary files)
- Create a GitHub release and add the archive as asset
- Use the app store API to publish a new release of the app https://nextcloudappstore.readthedocs.io/en/latest/restapi.html#publish-a-new-app-release

You could implement the automated app publishing in any way you want, but as an example you could look at how it is done for the app Cospend: https://github.com/julien-nc/cospend-nc/blob/main/.github/workflows/release.yml
see documentation here on how to publish your app to the app store in general.

- ## Workflow Generico

NB: Per i dettagli è meglio consultare i passaggi riportati nei capitoli precedenti.

- **Scheletro dell'applicazione**

- Go to **https://apps.nextcloud.com/developer/apps/generate**
- Extract app.tar.gz
- Copy-paste the directory of your app in workspace/server/apps-extra
- Enable it by going to (Dashboard)->Apps->Your Apps->Find the app and click "Enable"

./appinfo/info.xml:
- Aggiusta min-version, per eventuali componenti specifici utilizzati, e max-version, per la versione di Nextcloud attualmente utilizzata.
- Il tag <navigation> può essere tolto se non si ha bisogno dell'icona sulla navigation bar
- Aggiungi il tag <settings> per definire, ad esempio, la sezione Admin

./img/:
Cartella contenente immagini e icone utilizzate nell'applicazione.
È importante avere entrambe le versioni light (app.svg) e dark (app-dark.svg) dell'icona.

./l10n/:
Cartella contenente le traduzioni dell'applicazione, che si possono automatizzare.

- **Definizione di Application.php ed endpoint per le API**

./lib/AppInfo/Application.php:
File che dichiara le principali funzionalità dell'app, lato server, caricato ogni volta che viene caricata una pagina Nextcloud;.

./appinfo/routes.php:
Mappa tutti gli URL esposti dall'applicazione a specifici metodi php che gestiranno le richieste:
controller, endpoint, tipo di richiesta

- **Creazione dei controller e template**

./lib/: Cartella contenente tutte le classi PHP dei componenti dell'app (Controller, Dashboard, Provider...)

./lib/Controller/:
Contiene tutti i controller delle route dove vengono specificati i metodi scatenati dalle richieste.
Questi metodi devono contenere solo la gestione di base e il risultato delle richieste: tutto il processing dei dati va delegato ai Service.

./lib/Service/:
Contiene tutte le classi che implementano il reale backend delle funzionalità dell'applicazione.

./templates/:
I template PHP sono utilizzati per renderizzare pagine personalizzate, anche attraverso i file Javascript, definiti in ./src/
Lo stile invece può essere modificato tramite il file ./css/main.css

- **Gestione Javascript**

- Definisci la versione di Node.js da utilizzare (in questo caso 16) e installa NPM con tutte le dipendenze necessarie (salvate in package.json e package-lock.json)

.eslintrc.js:
Definisce regole eslint, controllate prima di compilare il codice

- **Frontend**

./src/:
Contiene tutti gli script Javascript
Il main.js, ad esempio, monta il “top-component” di Vue, App.vue, da cui derivano tutti gli altri.

./src/views/:
App.vue al suo interno importa tutti i componenti e metodi principali

./src/components/:
Contiene i componenti Vue dell’applicazione, ognuno con template (HTML), script (metodi, proprietà e definizione dei componenti) e style (CSS).

I componenti definiti o quelli di default possono poi essere usati ovunque attraverso..
import + aggiunta componente nell’attributo components dentro <script> + html
(per ogni componente, quando viene utilizziamo, bisogna specificare le proprietà)

./src/components/icons/:
Per le icone dei componenti è possibile utilizzare Vue Material Design, che definisce tutte le icone come componenti, dei quali quindi sarà necessario conoscere solo il nome assieme alle proprietà modificabili (https://pictogrammers.com/library/mdi/).

- **Compilazione script**

webpack.js:
I source file Javascript vengono compilati tramite Webpack, che quindi dovrà essere correttamente predisposto a tale funzione.

Creati gli script che verranno effettivamente caricati nel front-end dell’app li si compila tramite…
npm run dev (+ fix missing dependencies/errors)

- - - - - - - - - - - -

- **Database**

./lib/Migration/:
L’implementazione di un DB richiede innanzitutto una migrazione iniziale per creare le tabelle, la quale viene definita in questa cartella.

“The migration file name follows a naming convention where you fill in the year, the month, and the day, and the other numbers for hours, minutes and seconds.“

La migrazione si triggera cambiando il numero di versione in ./appinfo/info.xml (tag <version>) e refreshando l’istanza di Nextcloud.

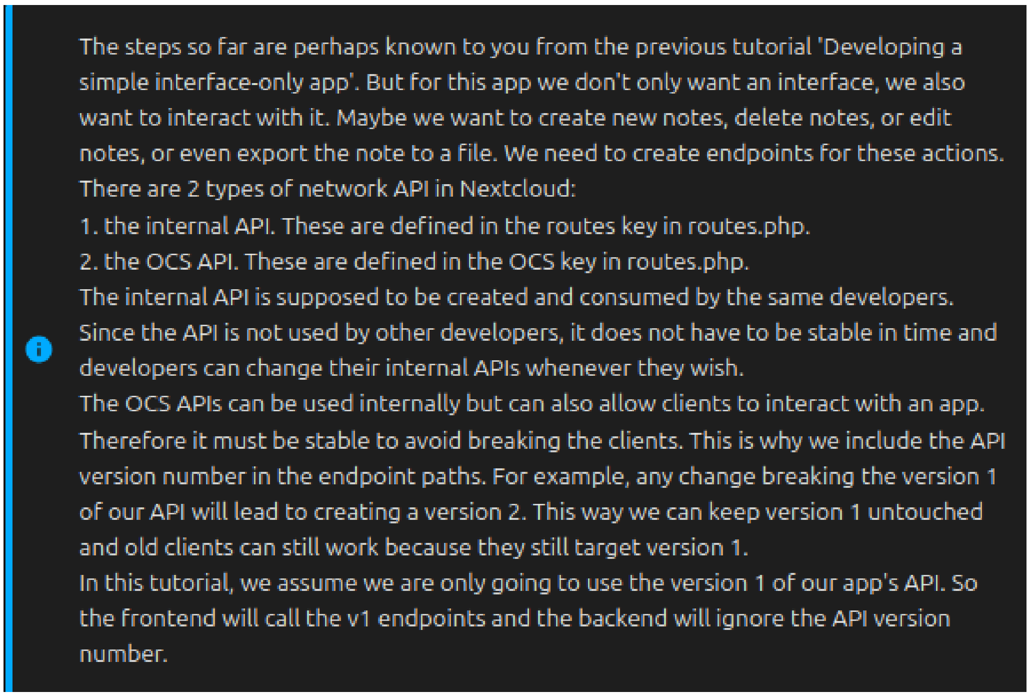
./lib/Db/:

Strato di astrazione per evitare di scrivere query SQL.

Contiene una classe per ogni tabella, ognuna con un unico mapper associato che definisce le operazioni che verranno eseguite nel db

Il mapper può essere utilizzato ovunque ma in ogni caso bisogna:
- Dichiarare l'utilizzo del mapper
- Dichiarare la classe del mapper
- Utilizzare il mapper

- **Interazione con Database**


The steps so far are perhaps known to you from the previous tutorial 'Developing a simple interface-only app'. But for this app we don't only want an interface, we also want to interact with it. Maybe we want to create new notes, delete notes, or edit notes, or even export the note to a file. We need to create endpoints for these actions.
There are 2 types of network API in Nextcloud:
1. the internal API. These are defined in the routes key in routes.php.
2. the OCS API. These are defined in the OCS key in routes.php.
The internal API is supposed to be created and consumed by the same developers. Since the API is not used by other developers, it does not have to be stable in time and developers can change their internal APIs whenever they wish.
The OCS APIs can be used internally but can also allow clients to interact with an app. Therefore it must be stable to avoid breaking the clients. This is why we include the API version number in the endpoint paths. For example, any change breaking the version 1 of our API will lead to creating a version 2. This way we can keep version 1 untouched and old clients can still work because they still target version 1.
In this tutorial, we assume we are only going to use the version 1 of our app's API. So the frontend will call the v1 endpoints and the backend will ignore the API version number.

All'interno di ./appinfo/routes.php quindi è possibile…
- "Restringere" la versione delle API concessa per effettuare richieste
- Dichiarare gli endpoint nel contenuto di 'ocs' (nell'array ritornato).

- **Integrazione servizi esterni tramite Smart Picker**

L'integrazione con servizi esterni può avvenire solo se sono presenti:
- URL per i singoli oggetti contenenti un identificativo
- API adatte a ciò che si desidera sviluppare
- API key, necessaria per poter accedere alle funzionalità esposte dal servizio

- Esegui `git pull origin master` e `git submodule update` per sviluppare in un environment aggiornato.

- Smart Picker può essere utilizzato nelle app Talk, Text, Collectives, Deck, Notes, and Mail. All'interno di workspace/server/apps-extra, quindi clona l'applicazione desiderata.

./lib/Reference/:

Contiene le classi PHP preposte alla <u>risoluzione dei link richiesti</u>

./lib/Listener/:

Contiene eventuali <u>event listener</u>

Per evitare di esporre l'API key si può pensare di integrarla direttamente in un pannello amministratore, creato appositamente:

./lib/Settings/:
- AdminSection.php: "registra" la classe per la sezione amministratore
- Admin.php: implementa la classe che otterrà l'API key e la fornirà al front-end

./templates/adminSettings.php:

Definisce il template per la classe appena implementata

./appinfo/info.xml:

Registriamo le impostazioni appena definite nel tag <settings>, subito prima del tag <dependencies>.

./src/adminSettings.js:

"Registra" i settings.

./src/components/AdminSettings.vue

Implementa i componenti Vue.js utilizzati nella sezione dell'amministratore.

Le icone sono definite in ./src/components/icons

- **Aggiunta test automatizzati**

Una volta pubblicata l'applicazione su GitHub i test verranno triggerati sulla base di ciò che è stato specificato in ./.github/workflows.

./.github/workflows/:

Cartella contenente tutti i file .yml, i quali mostrano le <u>azioni da eseguire al verificarsi di specifici eventi</u>.

./tests/:

Contiene file .xml che specificano la <u>classe di test eseguiti e ne mostrano il risultato</u>.

./tests/bootstrap.php:

File necessario per eseguire i test in un <u>ambiente configurato appropriatamente</u>.

./tests/unit/:

Cartella contenente la reale <u>implementazione dei test, ognuno formato da classe e mapper</u>.