Technical documentation

Stack

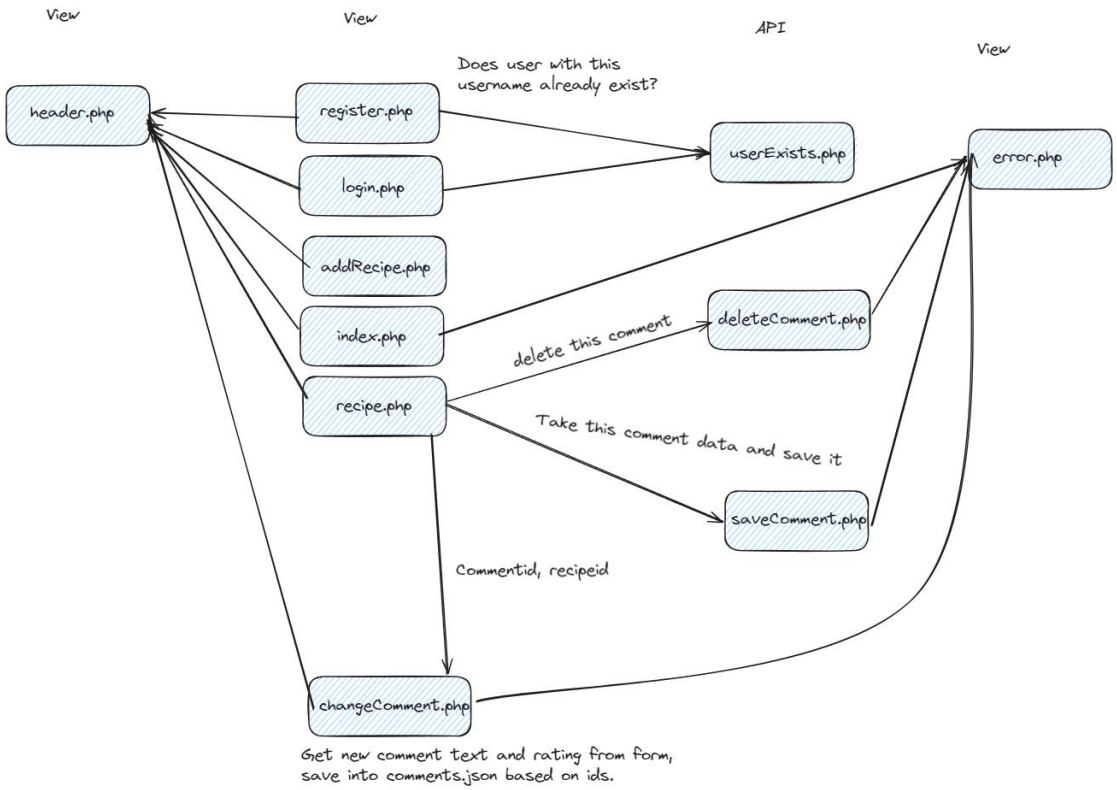
The website is built in vanilla HTML, CSS and Javascript on the front-end and vanilla PHP on the back-end. When developing, I used Apache as a web server. The website data isn't stored in a regular database, but instead in three JSON files stored on the server. Images are stored in separate files. The code is written mostly procedurally and it is not using MVC. While it would be a good design-pattern for the project, I only learned how to use it after doing the project.

File structure

The project contains multiple files and folders. The docs folder includes documentation (including this file), the css and js folders contain CSS and Javascript, which is then imported on individual pages. The CSS file name usually matches the file it's imported in, with some exception, for example, all pages have a header, and so I imported header CSS everywhere, and form.css is shared among multiple forms (login, register etc.). The images folder contains the cookbook.svg file which I use as an icon for the website, and it also contains all images uploaded by users. The files that aren't in folders are either PHP endpoints, JSON files for data, supplementary files like readme.md and the licence, and the .htaccess file, which I use to forbidden users to directly view any json files on the server and any dotfiles on the server.

Where to look first?

As a newcomer to the project, I recommend you first check out header.php and register.php files. header.php is a file that is included in every page and serves as the header, as the name suggests. Register.php is a page that is used for registering new users and shows you the type of form html and php validation that is present everywhere in the project. The PHP code checks if all needed fields are set, validates all read values in the POST superglobal, if they are incorrect, sends back the form prefilled, if they are correct, it creates a new user in users.json, creates a session and redirects to index.php, so the main page.



Endpoints structure – HTTP diagram

This is the schema of HTTP communication of my app. You can see 5 main files which I classified as views. These aren't just views, but they also handle their own form data. Most files access error and header, but that's not important. deleteComment and saveComment are classified as API, because they make changes to the server, without giving anything visual to the user. changeComment.php is kind of a middle ground. It gets a POST request with commentID and recipeID, puts it into hidden inputs, gets new contents of the comment from the user through a form, then handles it yourself and uses the ID data from before the change to the comment.

Session schema

Every logged-in user has three values saved in the session. bool [„loggedin“] which is a fast way to check if they are logged in, and string [„username“], which is their username. They also have string [„email“], but for users without email, this is just empty.

users.json schema

```
users.json{
    „johnDoe“:{
        „email“:“johndoe@email.cz“,
        „password“:“somehashgdfjghdjfghdjf“
    }
}
```

Additional users are additional keys. This has to mean that usernames are unique. Indeed, they are; this is checked as part of registration validation using AJAX in register.php. Of course, it is also checked on the server.

recipes.json schema

```
recipes.json {
    „randomidlike7b9c9a4f“: {
        „recipeName“:“food“,
        „description“:“how to make food (at least 20 characters)“,
        „ingredients“:[„apple“, „potato“],
        „author“: „John Doe“,
        „tags“:[„vegan“, „breakfast“]
    }
}
```

Each recipe has a unique id that serves as its key that is generated in addRecipe.php when you add a new recipe. This decision was made because nothing suitable in the recipe uniquely identifies it.

comments.json schema

```
comments.json {
    „recipeidlikeb9c9a“: [{
        „rating“:“5“,
        „comment“:“This recipe is good“,
        „author“: „John Doe“,
        „commentid“:“somerandomidlike674f2a6ed4fa“
    }, (here goes another comment)
    ]
}
```

The json file has keys that are identical to keys of recipes, so recipe ids. Each recipe has an array of objects as its value. The object has things like rating, comment etc. It also has a commentid that uniquely identifies it. This is very important for operations like changing and deleting comments.

What is each files job?

register.php

Job: Register users

If the user is logged in, redirect to index.php. If all fields in the form are set, validate each input. If something isn’t alright, send back a prefilled form with errors. This is done by echoing data in GET superglobal. If the data filled in the form is valid, hash the password, create an associative array conformant with users.json schema, read users.json, add the new object under the new key (username) and then save it to the filesystem. Using session superglobal, log in the user. Then, redirect to index.php so he can look at recipes.

login.php

Job: Login users

If the user is logged in, redirect to index.php. If all fields in the form are set, validate each input. If something isn’t alright, send back a prefilled form with errors. This is done by echoing data in the GET

superglobal. If the data filled in the form is valid, using session superglobal, log in the user. Then, redirect to index.php so he can look at recipes.

userExists.php

Job: Respond to requests that ask if username already exists in users.json.

This endpoint gets a GET request with a username included. (GET because its a read-only idempotent operation.) It needs to send back true or false based on if a user with this username already exists, using AJAX. Steps:
1. Read username from request.
2. Read users.json file.
3. Check if username is in users object.
4. respond with true, false, or an error in case someone deleted the database.

addRecipe.php

Job: Save a valid recipe to recipes.json.

If the user is logged in, redirect to index.php. If all fields in the form are set, validate each input. If something isn't alright, send back a prefilled form with errors. If valid, the goals are: read recipes.json, parse our recipe to an associative array conformant to the recipes.json schema, then update the recipes.json file on the filesystem, then redirect back so reloading the page doesn't send it again. The image in the form is saved into ./images/recipeID. The parsing is quite complicated, but well explained in the file itself.

index.php

Job: Showcase recipes on the website; let users sort them and filter them.
Recipes are loaded from recipes.json, each is given a key property which previously served as the key in recipes.json, this is because my parsing algorithm will destroy the keys and replace it with numbers, thus treating it like an array and easily iterating over it in a foreach loop. Only those recipes whose tags contain all tags wanted by the user are let through the filter function. Then, they are alphabetically sorted, or reverse alphabetically if the GET superglobal is set that way. In the view part, there is a foreach loop displaying all the data for each recipe. The most complex are tags (array of strings), which have to be translated using a dictionary and a map function, then displayed by joining the array into a string.

recipe.php

Job: Display finer details about a specific recipe and let users write and access comments.

The page itself doesn't do anything, but it uses a get parameter called recipeid for dynamic pagination, so each recipe.php?recipeid=someid represents an individual recipe. The PHP in this file is light, mostly just doing error handling and reading comments and recipe details from the json files, so they can be displayed below, which contains a lot of HTML. As shown in the diagram, it calls 3 API endpoints to handle comments that are explained below.

saveComment.php

Job: Take in information from the form in recipe.php and save comment to comments.json.

It reads rating, comment and recipeid from the form on recipe.php, because it is set as its action. If everything is valid and is set, (including session username, which we need), we can open comments.json file, and either create a new array for that key(recipeid) or push to an existing one.

Comment schema is described above. After the changes are written, user is redirected back to original recipe, since we have the recipeid. This handles post-redirect-get.

deleteComment.php

Job: Take recipeid and commentid from hidden inputs beside the delete button and remove comment from comments.json.

If recipeid and commentid are set, it reads the file, tries finding the comment, and if successful (ids match) and if the logged in user is the same as comment author, it removes the comment and writes changes to comments.json by unsetting that comment.

changeComment.php

Job: Use recipeid and commentid from hidden inputs, then let the user write a new comment and/or rating and use all four pieces of information to make a valid change to comments.json.
It is an actual page with html, not just an API endpoint. Hidden inputs with recipeid and commentid are echoed into the html form, which we need to do to preserve information from recipe.php. The user doesn't see that though, they need to fill in the other form elements to change rating and/or comment text. If everything is set correctly, we use the same finding algorithm for the comment as we did in deleteComment and we change the values in the object/assocArray to their new rating and comment. After everything is done, it is written to comments.json and user is taken back. In general, changeComment and deleteComment work almost the exact same way, except changeComment has the added complexity of also needing to take in information from the second form on the page itself.