Assigment 3 - PWA

Marc Lindegård Weller Bertelsen - Student nr 496141

21-11-2022

TODO-APP

Backend

Very short about the backend. It is a `Node.js` server, using the `express` framework to define the endpoints and `Mongoose` to access the `MongoDB` database.

Frontend Overview

The user is first greeted with a login screen. Enter a username and password, then press `Create New User` or `Login` if the user already exists, if the `Keep Loged In` box is checked, the user will be stored in `localStorage`. *note: username must be unique.*

Next up is the 3 lists of tasks: `Todo`, `Doing`, and `Done`.

Press the `+` next to `Todo`, to add a new task. *All tasks start on `Todo`.*

Press the `->` or `<-` to move a task to in the indicated direction.

Press `Edit` to enable the editing of a task, press `Save` to keep the changes.

Press `Delete` to remove a task.

Press `Logout` to logout and back to the login screen.

Run the app

There is a docker-compose included with project, so just run:

```
docker-compose up
```

Ports used:

- Database: `27017`
- Backend API: `3366`
- Frontend: `3000`

Routes

Although the `react-router-dom` is added to the project, and routing is added to the app, it was not nesseray as there is only 1 view/route.

Context

Since many different components require access to both the user and the tasts, 2 contexts are implemented in the app:

`/frontend/src/context/UserContext.ts`

`/frontend/src/context/TaskContext.ts`

MVVM

For the app MVVM is used to seperate the logic away from the ui

Models

The app implements 2 models:

/frontend/src/models/UserModel.ts

```
import { v4 as uuidv4} from "uuid";

export default class User {

    constructor(public _id: string,
                public username: string,
                public password: string) {
        this._id = (_id === "") ? uuidv4() : _id;
    }
}
```

/frontend/src/models/TaskModel.ts

```
import { v4 as uuidv4} from "uuid";

export enum TaskState {
    TODO = "TODO",
    DOING = "DOING",
    DONE = "DONE"
}

export default class Task {

    constructor(public _id: string,
                public user_id: string,
                public title: string,
                public description: string
                public state: TaskState = TaskState.TODO) {
        this._id = (!_id || _id.length === 0) ? uuidv4() : _id;
    }
}
```

ViewModels

The 2 models each have their own ViewModel to handel the logic specific to the model. Both ViewModels are implemented as singletons:

/frontend/src/viewmodels/UserViewModel.ts

```
import User from "../models/UserModel";
import Service from "../services/Service";

export class UserViewModel {
    private static instance: UserViewModel;
    private service: Service;

    private constructor() {
        this.service = new Service(process.env.REACT_APP_API_URL)
    }

    static getIntance(): UserViewModel {
        return this.instance === undefined ? this.instance = new UserViewModel() : this.instance;
    }

    retreiveUser(): User | undefined {
        ...
    }

    async login(username: string, password: string, stayLoggedIn: boolean): Promise<User | undefined> {
        ...
    }

    async createUser(username: string, password: string, stayLoggedIn: boolean): Promise<User | undefined> {
        ...
    }

    logout(): undefined {
        ...
    }

}

const userViewModel = () => {
    return UserViewModel.getIntance();
}

export default userViewModel;
```

/frontend/src/viewmodels/TaskViewModel.ts

The TaskViewModel uses a Map<string, Task>. where the key is the id of the task, to store the different tasks, so adding and removing a task can be done from the tasks id.

```
import Task from "../models/TaskModel";
import Service from "../services/Service";

export class TaskViewModel {
    private static instance: TaskViewModel;
    private service: Service;
    private taskMap: Map<string, Task>;

    private constructor() {
        this.service = new Service(process.env.REACT_APP_API_URL);
        this.taskMap = new Map();
    }

    static getIntance(): TaskViewModel {
        return this.instance ?? (this.instance = new TaskViewModel());
    }

    async createTask(task: Task): Promise<Task[]> {
        ...
    }

    async removeTask(task: Task): Promise<Task[]> {
        ...
    }

    async updateTask(task: Task): Promise<Task[]> {
        ...
    }

    async getAllTasks(user_id: string): Promise<Task[]> {
        ...
    }
}

const taskViewModel = () => {
    return TaskViewModel.getIntance();
}

export default taskViewModel;
```

View

The only view in the app is /frontend/src/views/Dashboard.tsx. Both the components /frontend/src/components/LoginForm.tsx and /frontend/src/components/Header.tsx, handles interactions with the UserViewModel. Interaction with the TaskViewModel is done with /frontend/src/components/TaskForm.tsxand /frontend/src/components/TaskCard.tsx

I decided to do all the css myself, for no good reason other than i wanted to.

All the components are loaded with lazy, as they are needed.

/frontend/src/App.tsx uses useEffect to check if a user was saved from last session:

```
useEffect(() => {
    setUser(userViewModel().retreiveUser());
}, []);
```

Then when the /frontend/src/views/Dashboard.tsx is mounted it checks if a user was found if it was it get all the tasks for that user:

```
useEffect(() => {
    if (user) {
        taskViewModel().getAllTasks(user._id).then(results => {
            setTasks(results);
        });
    }
}, [user, setTasks]);
```

Service

Since both the ViewModels interact with the backend, the /frontend/src/services/Service.ts wraps the fetch api:

```
import IResult from "../interfaces/IResult";

export default class Service {

    private headers = {
        "Content-Type": "application/json",
        "api_key": `${process.env.REACT_APP_API_KEY}`,
    } as const;

    constructor(private api_url?: string) {
        if (!api_url) console.log("API url is undefined!");
    }

    public async GET(endpoint: string): Promise<IResult> {
        ...
    }

    public async POST(endpoint: string, body: any): Promise<IResult> {
        ...
    }

    public async PATCH(endpoint: string, body: any): Promise<IResult> {
        ...
    }

    public async DELETE(endpoint: string): Promise<IResult> {
        ...
    }
}
```

IResult

The return type for all the mothods in /frontend/src/services/Service.ts is a Promise<IResult>. Since different things are expected from different endpoints, the interface /frontend/src/interfaces/IResult.ts is used:

```
import Task from "../models/TaskModel";
import User from "../models/UserModel";

export default interface IResult {
    message: string;
    value: {
        task?: Task,
        tasks?: Task[],
        user?: User,
    }
}
```

It always contain a message and value, with potentialy differnt content.