Newsletter Electron App

Team 06: Mathias, Camille, Paula

1. Notifications :

1.1 Notifications when saving/deleting an article

In the main.ts file :

(Receiver in main process)

```
ipcMain.handle('show-notification', (event, body: any) => {
  const notification = new Notification({
    title: body.title || 'Title missing',
    body: body.message || 'Body missing',
    silent: true,
  })
  notification.on('click', () => {
    event.sender.send('notification-clicked')
  })
  console.log(notification)
  notification.show()
})
```

In the electron.service.ts file :

```
sendNotification(body: { title: string; message: string; callback?: () => void }) {
    if (this.isElectron) {
        this.ipcRenderer.invoke('show-notification', {
            title: body.title,
            message: body.message,
            callbackEvent: 'notification-clicked-answer',
        })

        this.ipcRenderer.on('notification-clicked', () => {
            if (body.callback) {
                body.callback()
            }
        })
    } else {
        return Promise.reject('Not running in Electron environment')
    }
}
```

- Notify users when an article is successfully created:

In the article-edition.component.ts file :

We use the method sendNotification from an object of type ElectronService (electSvr) to notify to the user that the article has been saved correctly. It's use in our method save() which is called when the user click on the button "save" in the edition page of an article.

```
this.electSvr.sendNotification({
  title: `The article : ${this.article.title} has been saved`,
  message:'You will be redirect to the main page',
  callback: () => {
    console.log(`The article "${this.article.title}" has been saved.`);
  },
});
```

- Notify users when an article is successfully deleted.

In the article-list.component.ts file :

We use the method sendNotification from an object of type ElectronService (electSvr) to notify the user that the article has been deleted correctly. It's used in our method delete(article: Article) which is called when the user clicks on the button "delete" in the main page (where we can see all articles). This is visible only for logged in users.

```
this.electSvr.sendNotification({
  title: `The article : ${article.title} has been deleted`,
  message:'The operation was successful.',
  callback: () => {
    console.log(`The article "${article.title}" has been deleted.`);
  },
});
```

1.2 Notifications when editing an article with an error

In the file article-edition.component.css file we define this class that will be used to show to the user the error (in red).

```
.error-highlight {
  border: 2px solid red;
  border-radius: 4px;
  background-color: #ffe6e6; /* Light red background */
}
```

```
// Validate required fields
const errors: { field: string; message: string }[] = [];

if (!this.article.title?.trim()) {
  errors.push({ field: 'title', message: 'The title is required.' });
}
if (!this.article.subtitle?.trim()) {
  errors.push({ field: 'subtitle', message: 'The subtitle is required.' });
}
if (!this.article.abstract?.trim()) {
  errors.push({ field: 'abstract', message: 'The abstract is required.' });
}
if (!this.article.category?.trim()) {
  errors.push({ field: 'category', message: 'The category is required.' });
}

// Remove error highlighting from all fields
const allFields = ['title', 'subtitle', 'abstract', 'category'];
allFields.forEach((field) => {
  const element = document.getElementById(field);
  if (element) {
    element.classList.remove('error-highlight');
  }
});
```

We use a variable “errors” to store an error (if there is any).

```
// If there are validation errors, highlight fields and scroll to the first error
if (errors.length > 0) {
  errors.forEach((error) => {
    const errorElement = document.getElementById(error.field);
    console.log(`Error on field: ${error.field}, Found element:`, errorElement);
    if (errorElement) {
      // Add red highlight to fields with errors
      errorElement.classList.add('error-highlight');
    }
  });

  const firstError = errors[0];
  const firstErrorElement = document.getElementById(firstError.field);
  if (firstErrorElement) {
    // Scroll smoothly to the first field with an error
    firstErrorElement.scrollIntoView({ behavior: 'smooth', block: 'center' });
    firstErrorElement.focus(); // Focus on the field for the user's convenience
  }

  // Send notifications for each validation error
  errors.forEach((error) => {
    this.electSvr.sendNotification({
      title: 'Validation Error',
      message: error.message,
    });
  });

  return; // Stop execution if there are validation errors
}
```

We use the scrollIntoView function to go to the error so we can show it to the user with a notification.

2. Data Management: Import/Export

In the main.ts file :

(Receiver in main process)

- **For the export :**

```
ipcMain.handle('export-article-json', (event, articleJson: string) => {
  try {
    // Parse the JSON to access the article title
    const article = JSON.parse(articleJson);
    let articleTitle = article.title;

    /* Clean the title to use it as a valid file name
       replace invalid characters with '_'
    */
    articleTitle = articleTitle.replace(/[^a-zA-Z0-9-_]/g, '_');

    const exportPath = path.join(os.homedir(), 'Desktop', `${articleTitle}.json`);
    fs.writeFileSync(exportPath, articleJson, 'utf8');

    dialog.showMessageBox({
      type: 'info',
      title: 'Article Exported',
      message: 'Article exported to: ' + exportPath,
      buttons: ['OK'],
    });

    return { success: true, path: exportPath };
  } catch (error: any) {
    return { success: false, error: error.message };
  }
});
```

- **For the import :**

```
ipcMain.handle('import-text', async (event) => {
  try {
    // Show the file dialog to choose a .txt file
    const result = await dialog.showOpenDialog({
      properties: ['openFile'],
      filters: [{ name: 'Text Files', extensions: ['txt'] }],
    })

    // Check if the user selected a file
    if (result.canceled || result.filePaths.length === 0) {
      return { success: false, error: 'No file selected' }
    }

    const filePath = result.filePaths[0]

    // Read the file content as text
    const fileContent = fs.readFileSync(filePath, 'utf8')

    // Return the file content
    return { success: true, content: fileContent }
  } catch (error: any) {
    return { success: false, error: error.message }
  }
})
```

In the electron.service.ts file :

- **For the export :**

```
exportArticleAsJson(articleJson: string) {
    if (this.isElectron) {
        return this.ipcRenderer.invoke('export-article-json', articleJson)
    }
  }
```

- **For the import :**

```
async importArticle(): Promise<string> {
    const result: string = await this.ipcRenderer.invoke('import-text')
    return result
}
```

In the article-list.component.html file :

We show the user a button in the main page “Export as JSON” if he is logged in. When he clicks on this button a file with the title as name will be downloaded on his desktop.

```
<a *ngIf="loginSrv.isLogged()" class="btn btn-dark" (click)="exportArticleToJson(article)">Export as JSON</a>
```

In the article-details.component.html file :

We show the user a button at the end to “Export as JSON” if he is logged in, too. The behavior is the same as in the list.

```
<button *ngIf="loginSvr.isLogged()" class="btn btn-dark btn-success mx-2" (click)="exportArticleToJson(article)">Export as JSON</button>
```

In the article-list.component.ts and in the article-details.component.ts files:

We have defined the function to export articles as JSON.

```
  exportArticleToJson(article: Article): void {
    if (!article) {
      console.error('Article is undefined');
      alert("There was an error to export the article")
      return;
    }
    // We change the format of article to JSON and use the electron Service
    const articleJson = JSON.stringify(article);
    this.electSrv.exportArticleAsJson(articleJson);
  }
```

- **For the import part**, we display this option only when a user wants to create a new article or when he wants to modify an article.

In the article-edition.component.html and in the article-creation.component.html files, we call the <app-filesystem> and gives the article

```
  <div class="mt-4 mb-4">
    <app-filesystem [article]="article"></app-filesystem>
</div>
```

Indeed, we will use the article to set the data in the form and so the user can retrieve the article he just imported. So in the filesystem.component.ts file, we use the Input we got before.

```
export class FilesystemComponent {
  /*
  The article edition page gives to the file system
  the article the user wants to import or export
  */
  @Input() article: any;
```

The function “setArtcleData(data: any)” will be used in the final function change when the user clicks on the import button. When, we get the data (the JSON file) and when this data is parsed we will call this function (see the second screenshot below)

```
// Function to assign JSON data to form fields
setArticleData(data: any) {
  this.article.title = data.title;
  this.article.subtitle = data.subtitle;
  this.article.category = data.category;
  this.article.abstract = data.abstract;
  this.article.body = data.body;
  if (data.image_data) {
    this.article.thumbnail_image = '' + data.image_data;
  }
}
```

```
onFileChange(event: any) {
  const file = event.target.files[0];
  if (file) {
    const fileNameElement = document.getElementById('fileName') as HTMLElement;
    fileNameElement.innerText = file.name;
  }
  if (file && file.type === 'application/json') {
    const reader = new FileReader();
    reader.onload = (e: any) => {
      const content = e.target.result;
      try {
        const data = JSON.parse(content);
        this.setArticleData(data);
      } catch (error) {
        this.importError = 'Error reading JSON file';
      }
    };
    reader.readAsText(file);
  } else {
    this.importError = 'Please upload a valid JSON file';
  }
}
```

In the filesystem.component.html file we have the import design with the call of the function onFileChange.

```html
<div class="filesystem-container">
    <h4>Import Article</h4>
        <div class="custom-file-input">
          <label for="fileInput" class="file-button">Choose File</label>
          <input type="file" id="fileInput" (change)="onFileChange($event)" />
          <span id="fileName">No file chosen</span>
        </div>
</div>
```

3. Enhance Desktop Look and Feel

3.1 Draggable Header

In the article-edition.component.css, article-creation.component.scss, article-details.component.css, and article-list.component.css files, we defined the draggable header with some options, but the most important thing is the first line.

```css
.draggable-header {
  -webkit-app-region: drag;
  background-color: #333;
  color: white;
  height: 40px;
  display: flex;
  align-items: center;
  justify-content: space-between;
  padding: 0 10px;
  user-select: none;
  }
```

Then, we can call it in the article-edition.component.html, article-creation.component.html, article-details.component.html, and article-list.component.html files adding the following line:

```html
<div class="mt-4 draggable-header" style="background-color: #f3f3f3;">
```

3.2 Disable text selection for all text elements.

In the article-edition.component.css, article-creation.component.scss, article-details.component.css, and article-list.component.css files, we defined the no-select class to prevent users from selecting text in any component where this class is applied.

```css
.no-select {
  user-select: none;
}
```

Then, we can call it in the article-edition.component.html, article-creation.component.html, article-details.component.html, and article-list.component.html files adding the following line:

```html
<main class="pt-2 flex-fill flex-column d-flex no-select">
```

3.3 Mouse cursor with default look only

In the article-edition.component.css, article-creation.component.scss, article-details.component.css, and article-list.component.css files, we turned off mouse cursor when hovering links so it only uses the default look:

```
a {
  cursor: 'default';
}
```

3.4 Hide window until app loads

In the main.ts file, we added this to avoid showing the window until app has finished loading:

```
win.once('ready-to-show', () => {
  if(win){
    win.show();
  }
});
```

4. Session Persistence :

We used the library electron-store (install the library with : npm install electron-store).

Then import the library in the app/main.ts file and create the getValue() and storeValue() functions :

```
const Store = require('electron-store');

const store = new Store();
```

```
ipcMain.handle('store-value', (event, key: string, value: string) => {
  store.delete(key);
  store.set(key, value);
  console.log(`Stored ${key} ${store.get(key)}`);
  return null;
});

ipcMain.handle('get-value', (event, key: string) => {

  const value = store.get(key);
  console.log(`Retrieved ${key}: ${value}`);
  return { path: key, data: value };
});
```

Let Angular access the function by including them in core/electron.service.ts

```
    storeValue(key: string, value: string) {
        if (this.ipcRenderer) {
            this.ipcRenderer.invoke('store-value', key, value)
        } else {
            localStorage.setItem(key, value)
        }
        return null
    }

    async getValue(key: string): Promise<{ path: string; data: string }> {
        let value
        if (this.ipcRenderer) {
            value = await this.ipcRenderer?.invoke('get-value', key)
        } else {
            value = localStorage.getItem(key)
        }
        return value
    }
```

Use them in login.service.ts :

While logging :

```
login(name: string, pwd: string): Observable<User> {
  const usereq = new HttpParams()
    .set('username', name)
    .set('passwd', pwd);

  return this.http.post<User>(this.loginUrl, usereq).pipe(
    tap(user => {
      this.user = user;
      this.newsSrv.setUserApiKey(user.apikey);
      this.electronService.storeValue("userNameToken", user.username);
      this.electronService.storeValue("userPwdToken", pwd);
    })
  );
}
```

To initialize login service :

```
async initializeLogin(): Promise<[string, string]> {
  try {
    // Get tokens from Electron store
    const userToken = await this.electronService.getValue('userNameToken');
    const pwdToken = await this.electronService.getValue('userPwdToken');

    // Check if both tokens are present
    if (userToken?.data && pwdToken?.data) {
      const userName = userToken.data;
      const password = pwdToken.data;

      // Optionally, you can validate the tokens or check expiration here
      console.log('Tokens found, logging in user...');
      return [userName, password];
    }

    console.log('No token found, prompting user to log in...');
    return ['', '']; // No tokens found, user needs to log in
  } catch (error) {
    console.error('Error during token initialization', error);
    return  ['', '']; // Return false in case of any error
  }
}
```

Then update the login.component.ts to actually initialize login service :

```
async ngOnInit() {
    const userId : [string, string] = await this.loginSrv.initializeLogin();

    this.loginSrv.login(userId[0], userId[1]).subscribe({
      next: (user: User) => {
        this.user = user;
        this.isLogged = this.loginSrv.isLogged();
      },
      })
  }
```