VILNIUS UNIVERSITY

KAUNAS FACULTY

INSTITUTE OF SOCIAL SCIENCES AND APPLIED INFORMATICS

Study program Information Systems and Cyber Security

**WWW Development Technologies**

**Uletter**

**Kaunas 2024**

**Vladyslav Nester, Oleksandr Lanskyi**

Contents

[Table of figures 2](#_Toc184703979)

[Table of tables 3](#_Toc184703980)

[Research and Analysis 4](#_Toc184703981)

[What requirements of the consumers are you trying to fulfill 5](#_Toc184703982)

[What are the non-negotiable requirements for the website? 5](#_Toc184703983)

[Design 6](#_Toc184703984)

[The selection of a technology stack 7](#_Toc184703985)

[Content 7](#_Toc184703986)

[Content Creation Content Strategy 8](#_Toc184703987)

[Communication Channels 9](#_Toc184703988)

[Content Samples 10](#_Toc184703989)

[Functional Requirements: 11](#_Toc184703990)

[Non-Functional Requirements: 11](#_Toc184703991)

[Activity diagram 12](#_Toc184703992)

[Use case diagram 13](#_Toc184703993)

[life cycle methodology 14](#_Toc184703994)

[Step-by-Step usability Guide 15](#_Toc184703995)

[Entity diagram 16](#_Toc184703996)

[Tests 16](#_Toc184703997)

[Backend 16](#_Toc184703998)

[Users 17](#_Toc184703999)

[Style\_personalization 20](#_Toc184704000)

[Personalization 24](#_Toc184704001)

[Frontend 25](#_Toc184704002)

[auth.js API Tests 25](#_Toc184704003)

[axiosInstance.js Tests 26](#_Toc184704004)

[profile.js Tests 27](#_Toc184704005)

[Conclusion 30](#_Toc184704006)

# Table of figures

[Figure 1 The slogan 10](#_Toc184703969)

[Figure 2 About page content 10](#_Toc184703970)

[Figure 3 Activity diagram 12](#_Toc184703971)

[Figure 4 Use case diagram 13](#_Toc184703972)

[Figure 5 Sequence diagram 14](#_Toc184703973)

[Figure 6 Step-by-Step usability Guide 15](#_Toc184703974)

[Figure 7 Entity diagram 16](#_Toc184703975)

[Figure 8 test auth.test.js 26](#_Toc184703976)

[Figure 9 Test axiosInstance.test.js 27](#_Toc184703977)

[Figure 10 Test profile.test.js 28](#_Toc184703978)

# Table of tables

[Table 1 7](#_Toc184703961)

# Research and Analysis

**Purpose of the website**

This application is designed to assist students who are new to university, as well as school students, in writing formal letters. It aims to simplify the process of crafting well-structured, professional letters for various situations. For example, a university student may need to write a letter to a professor or administrative staff, and a school student might need to address a letter to a teacher. The app will help users create these letters with the proper tone, structure, and formality. Additionally, it can assist users in drafting letters to supervisors, mentors, or other formal contacts, ensuring that the content meets expected standards and effectively conveys the intended message.

**Type of website app**

The website will be a commercial, service-oriented application designed to assist users with writing formal letters. At its initial stages, the app will be free to use, providing users with unlimited access. However, as the platform develops, a limit on the number of weekly requests will be introduced. Users who need additional requests will have the option to purchase a subscription or pay a fee for increased access.

**Target audience**

The target audience for this application includes students, particularly those new to university, as well as school students and young professionals. These users often need assistance with writing formal letters to teachers, professors, supervisors, or administrative staff. The app is also suitable for anyone seeking to create properly formatted and professional letters for academic, work, or personal purposes.

**Is there adequate demand in the market**

Yes, there is a significant demand for such AI applications in the market. Many students and young professionals face difficulties in writing formal letters, especially those who are moving to a university environment or starting their career. Writing professional and well-structured letters is a skill that not everyone has, and many are looking for help with formal communication.

**Why do users need this website**

Users need this website because it provides an easy and efficient way to create formal, professional letters, especially for those who may lack experience or confidence in writing.

What requirements of the consumers are you trying to fulfill

1. Provide a platform that simplifies letter writing for users, ensuring ease of use and accessibility.
2. Deliver tools for rewriting, generating, and customizing letters based on user preferences and styles.
3. Offer guidance and support to help users understand and effectively use the application.
4. Ensure personalized experiences through features like saved styles, message history, and editing options.
5. Cater to students, professionals, and anyone in need of formal communication assistance.

What are the non-negotiable requirements for the website?

1. A user-friendly and intuitive interface that allows seamless navigation and interaction.
2. High performance and responsiveness across devices, ensuring a smooth user experience.
3. Secure handling of user data, including personal information and writing history.
4. Clear and engaging content that explains functionality and supports user understanding.
5. Integration of core features such as message editing, style customization, and feedback collection.
6. Reliable technical infrastructure, ensuring minimal downtime and fast load times.

**Team expectations for look and feel**

The website will have a modern and clean design. The color palette will be neutral and professional, using shades of blue, white, and gray. The typography will feature clear and accessible fonts. The layout will be well-organized with intuitive navigation.

**How and what will the website generate according to the goals?**

The site will generate personalized formal letters tailored to the specific needs and context of the user. This will be done using integration with the ChatGPT API, which processes user input and pre-defined behavior patterns in the code, and the system will also tailor the response to each user using personalization algorithms.

**What features will the user expect from such an application and what are some features you wish to include?**

The user expects a simple and functional tool that simplifies the letter-writing process and ensures high-quality results. Our team would like to include the following features. A preview function to check the letter's format. Integration with downloadable formats such as PDF and Word to enable convenient sharing of letters. Examples and templates for various types of letters to guide users.

Design

In the future, it is planned to adhere to a strategy that includes all aspects of the website, such as design, technology, content, and marketing. In the process of implementation, it is planned to define the overall aesthetic of the website, focusing on user experience (UX) and user interface (UI). A consistent color scheme, typography, and layout style are planned to align with the branding. In the design, it is planned to include responsive features to ensure compatibility across various devices.

The selection of a technology stack

Table 1

Selection of a technology stack

|  |  |
| --- | --- |
| **Task** | **Chosen Tools** |
| |  | | --- | | Backend Development | | |  | | --- | | Python, Django | |
| |  | | --- | | Frontend Development | | |  | | --- | | React, Tailwind CSS, JavaScript | |
| |  | | --- | | API Integration | | Django REST Framework (DRF) |
| |  | | --- | | Database Management | | |  |  | | --- | --- | | SQLite, Django ORM | | |  | |
| Styling and Design | Tailwind CSS |

Content

The Contact page will include phone numbers, Facebook and Twitter links, and an email address for contacting support. Support details will also be provided, including business hours (Mon-Fri, 9:00-17:00) and estimated response time (within 24 hours on weekdays).   
Additionally, there will be a feedback form for user suggestions and queries.

The Home page will have a brief introduction to ULetter and its features. The page will show how the app simplifies the email writing process, include user testimonials and a call-to-action button to start using the app.

On the Tools page, users will find descriptions of additional functionality as well as links to related pages. Among the functionality will be the application of preferred writing styles and a direct link to the Edit Messages page for creating and customizing emails.

On the Edit Messages page, users will be able to access the core functionality of the app, set requirements for emails and customize generated emails. Users will be able to edit text, adjust tone and download the final version in PDF or Word format.

On the Manage Style page, users will be able to create and save custom styles for emails. They will be able to customize their preferred format, tone or structure to ensure consistency across multiple documents. The page will also offer options for applying saved styles to new emails.

The About page will include information about ULetter's mission and goals, a brief history of the project's creation, and a description of the team working on the platform.

The Profile page will display the user's photo and personal information at the top. Below that will be a section displaying the history of requests and usage of the app. In addition, there will be a text input section where the user can specify their preferences to customize the app's functionality to their needs.  
  
Designing

During the design process, our team decided not to use pre-prepared drafts of the pages or application. This decision was made collectively, as it was agreed that a more productive and satisfactory approach for all team members would be to allow each developer to implement their preferred page designs. These designs would then be evaluated collectively to determine their relevance and suitability.

One of the key motivations for this decision was the desire to preserve the creative input of each developer, allowing them to explore innovative ideas during the creation process. This approach ensures that the design process remains dynamic and fosters originality while maintaining team collaboration and alignment.

# Content Creation Content Strategy

Apart from the core functionality, the textual and visual content on the pages will primarily serve an introductory purpose. It will provide explanations of the application's functionality, general information, and all the necessary details to assist users in understanding and navigating the application effectively. The content will be designed to be clear, concise, and user-focused, ensuring that users can quickly grasp the features and benefits of the platform.

Communication Channels

Our communication channels offer users many ways to contact us. Users can visit us in person at our address, send a message via Facebook or Twitter, phone, email or use the feedback option available on our website. Feedback sent via the website is stored in a database, which is managed by a member of staff who is responsible for rectifying complaints and responding to user queries. This integrated approach was chosen to allow users to choose their preferred method of contact, improving accessibility and responsiveness.

Content Samples

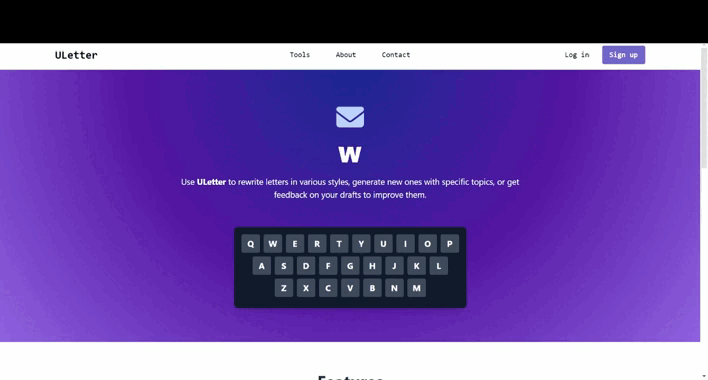


Figure 1 The slogan

The slogan on the website is presented in an interactive and visually engaging manner, designed to grab the user’s attention and focus them on the platform’s core purpose. The dynamic design and vibrant colors ensure the message stands out, making the user immediately interested in exploring the features further.

A close-up of a person typing on a computer

Description automatically generated

Figure 2 About page content

The user can also find content dedicated to the project that explains its design and provides detailed usage guidelines. This was added to ensure that users are well informed about the features and functions of the platform, making it easier to use and more efficient.

# Functional Requirements:

1. The website must allow users to rewrite letters in various styles (e.g., formal, concise, or slang).
2. Users should be able to create letters from scratch by specifying the subject and tone.
3. The system must provide a message editing tool as the primary functionality for letter customization.
4. A personalized style management feature must allow users to save and apply their preferred styles.
5. The profile page must display user details, message history, and all-time statistics.
6. The platform should allow users to upload drafts and receive actionable feedback for improvements.
7. Contact information and a feedback submission system must be available on the Contacts page.
8. The Tools page must link users to additional features such as style management and the message editing page.
9. The system must validate the strength of user passwords during registration or password updates and provide warnings if the password is weak.
10. The application must include an authentication system, allowing users to log in, sign up, and manage their accounts.

# Non-Functional Requirements:

1. The website must have a clean, intuitive, and responsive user interface to ensure ease of navigation.
2. The system must support fast performance, with page load times under 2 seconds for most interactions.
3. Data security measures must be in place to protect user information, adhering to GDPR or similar standards.
4. The website must be compatible across all major browsers (e.g., Chrome, Firefox, Edge, Safari).
5. Scalability must be ensured to handle a growing number of users without performance degradation.
6. The design must align with branding guidelines, maintaining a professional and visually appealing layout.
7. The system should provide 90,3% uptime reliability for uninterrupted user access.
8. Feedback submission must include a database with an efficient mechanism for tracking and resolving complaints.
9. All functionalities must include tooltips or guides for first-time users to ensure usability.
10. The platform must support seamless integration with external APIs (e.g., ChatGPT for content generation).

# Activity diagram



Figure 3 Activity diagram

# **Use case diagram**

A diagram of a diagram

Description automatically generated

Figure 4 Use case diagram

# Sequence diagram



Figure 5 Sequence diagram

# life cycle methodology

We chose Agile methodology because it aligns with our workflow and team dynamics. Agile allows parallel development, enabling team members to work on different components simultaneously. Frequent code integrations support incremental progress and quick issue identification. Iterative testing ensures continuous quality improvements throughout the development process. Collaboration is enhanced as team members contribute in areas where they excel. Flexible schedules allow each member to work according to their availability. Diverse starting points make it possible to begin work on different parts of the project without dependencies. Models like Waterfall or V-Model were unsuitable due to their rigid sequential structure, which does not support parallel tasks or iterative progress effectively.

# Step-by-Step usability Guide

A diagram of a company

Description automatically generated

Figure 6 Step-by-Step usability Guide

# Entity diagram

A screenshot of a computer

Description automatically generated

Figure 7 Entity diagram

# Tests

## Backend

Text\_processing

Test\_OpenAIHelper.py

##### test\_generate\_system\_content\_valid

**Description**: Tests if the *generate\_system\_content* method generates the correct system content for valid inputs.

**Setup**: Calls *OpenAIHelper.generate\_system\_content* with valid arguments: *"professional"* tone, *"email"* type, and *"edit"* purpose.

**Test Logic**: Verifies that the generated content includes the words "formal" and "email".

**Expected Behavior**: The method should generate a content string that contains references to both the specified tone and type.

##### test\_generate\_system\_content\_invalid

**Description**: Tests if the *generate\_system\_content* method raises a *ValueError* for invalid inputs.

**Setup**: Calls *OpenAIHelper.generate\_system\_content* with an invalid tone ("invalid-tone") and valid type and purpose.

**Expected Behavior**: The method should raise a *ValueError* to indicate unsupported tone values.

### Users

#### Test\_LoginView.py

##### test\_user\_login\_successful

**Description**: Tests if a registered user can log in successfully with valid credentials.

**Setup**: A test user is created in the *setUp* method with valid credentials.

The login endpoint (*'login'*) is tested with a valid payload containing the correct username and password.

**Expected Behavior**: The API should return a **201 Created** status code.

The response should include access and refresh tokens, confirming successful authentication.

##### test\_user\_login\_invalid\_credentials

**Description**: Tests the behavior of the Login View when invalid credentials are provided.

**Setup**: The login endpoint (*'login'*) is tested with an invalid payload containing the correct username but an incorrect password.

**Expected Behavior**: The API should return a **400 Bad Request** status code.

The response should include an error message in the *non\_field\_errors* field, indicating invalid credentials.

#### Test\_SignupView.py

test\_user\_registration\_successful

**Description**: Tests if a new user can register successfully with valid input data.

**Setup**: The signup endpoint (*'signup'*) is tested with a valid payload containing all required fields, including matching password and *password\_confirm*.

**Expected Behavior**: The API should return a **201 Created** status code.

The response should include access and refresh tokens, confirming the successful creation of a new user account.

test\_user\_registration\_password\_mismatch

**Description**: Tests the behavior of the Signup View when the password and *password\_confirm* fields do not match.

**Setup**: The signup endpoint (*'signup'*) is tested with an invalid payload where password and password\_confirm have different values.

**Expected Behavior**: The API should return a **400 Bad Request** status code.

The response should include a validation error for the *password\_confirm* field, indicating the mismatch.

#### Test\_UserAccountDeactivationView.py

test\_account\_deactivation

**Description**: Tests if an authenticated user can deactivate their account successfully.

**Setup**: A test user is created in the *setUp* method.

A *JWT access\_token* and *refresh\_token* are generated for the user using *RefreshToken.for\_user*. The *HTTP\_AUTHORIZATION* header is set with the *access\_token* to authenticate the user. A valid payload containing the *refresh\_token* is prepared.

**Expected Behavior**: The API should return a **200 OK** status code.

The user's *is\_active* status should be set to False, effectively deactivating the account.

#### Test\_UserProfileChangePasswordView.py

##### test\_password\_change\_successful

**Description**: Tests if an authenticated user can change their password successfully.

**Setup**: A test user is created in the *setUp* method. A *JWT access\_token* is generated using *RefreshToken.for\_user*, and the *HTTP\_AUTHORIZATION* header is set with the token to authenticate the user.

A valid payload with matching *new\_password* and *password\_confirm* fields is prepared.

**Expected Behavior**: The API should return a **200 OK** status code.

The response should include a success message: "Password changed successfully."

##### test\_password\_change\_mismatch

**Description**: Tests if the view correctly handles cases where the *new\_password* and *password\_confirm* fields do not match.

**Setup**: Uses an invalid payload where *new\_password* and *password\_confirm* do not match.

**Expected Behavior**: The API should return a **400 Bad Request** status code.

The response should include an error for the *password\_confirm* field, indicating the mismatch.

#### Test\_UserProfileView.py

##### test\_password\_change\_successful

**Description**: Tests if an authenticated user can change their password successfully.

**Setup**: A test user is created in the *setUp* method.

A *JWT access\_token* is generated using *RefreshToken.for\_user*, and the HTTP\_AUTHORIZATION header is set with the token to authenticate the user.

A valid payload with matching *new\_password* and *password\_confirm* fields is prepared.

**Expected Behavior**: The API should return a **200 OK** status code.

The response should include a success message: "Password changed successfully."

##### test\_password\_change\_mismatch

**Description**: Tests if the view correctly handles cases where the *new\_password* and *password\_confirm* fields do not match.

**Setup**: Uses an invalid payload where *new\_password* and *password\_confirm* do not match.

**Test Logic**: Sends a *PUT* request to the password-change endpoint with the invalid payload.

**Expected Behavior**: The API should return a **400 Bad Request** status code.

The response should include an error for the *password\_confirm* field, indicating the mismatch.

### Style\_personalization

#### Test\_model.py

#### StyleModelTest

**Setup**: Creates a user and a *Style* instance for testing attributes like *name*, *category*, and string representation (*\_\_str\_\_*).

##### test\_style\_creation:

**Description**: Tests if the Style model creates an object with valid data.

**Expected Behavior**: A Style object should be created with the specified fields.

##### test\_style\_str\_method:

**Description**: Tests the \_\_str\_\_ method of the Style model.

**Expected Behavior**: The \_\_str\_\_ method should return the name field or "Style <id>".

#### QuestionListModelTest

**Setup**: Creates a *QuestionList* instance for testing fields like *category* and string representation.

##### test\_question\_list\_creation:

**Description**: Tests if the *QuestionList* model creates an object with valid data.

**Expected Behavior**: A *QuestionList* object should be created with the specified category and question.

##### test\_question\_list\_str\_method:

**Description**: Tests the \_\_str\_\_ method of the *QuestionList* model.

**Expected Behavior**: The \_\_str\_\_ method should return a string in the format "<category>: <question>".

#### StyleQuestionsModelTest

##### test\_style\_question\_creation:

**Description**: Tests if the *StyleQuestions* model creates an object with valid data.

**Expected Behavior**: A *StyleQuestions* object should be created and linked to the Style object.

##### test\_style\_question\_str\_method:

**Description**: Tests the \_\_str\_\_ method of the *StyleQuestions* model.

**Expected Behavior**: The \_\_str\_\_ method should return a string in the format "Question for style <style name>".

#### Test\_serializer.py

#### StyleSerializerTest

**Setup**: Prepares valid and invalid payloads to test serialization and validation logic for the *StyleSerializer*.

##### test\_style\_serializer\_valid\_data:

**Description**: Tests if the *StyleSerializer* correctly validates and serializes valid input data.

**Expected Behavior**: The serializer should be valid and return the serialized data matching the input.

##### test\_style\_serializer\_invalid\_data:

**Description**: Tests if the *StyleSerializer* correctly handles invalid input data, such as missing required fields.

**Expected Behavior**: The serializer should be invalid and return appropriate validation error messages.

##### test\_style\_serializer\_missing\_questions:

**Description**: Tests if the *StyleSerializer* correctly validates that at least one question is required.

**Expected Behavior**: The serializer should return a validation error for missing questions.

#### StyleQuestionSerializerTest

**Setup**: Prepares payloads to test the validation of question-and-answer fields in *StyleQuestionSerializer*.

##### test\_style\_question\_serializer\_valid\_data:

**Description**: Tests if the *StyleQuestionSerializer* validates and serializes valid question and answer data.

**Expected Behavior**: The serializer should be valid and return the serialized data.

##### test\_style\_question\_serializer\_empty\_question:

**Description**: Tests if the *StyleQuestionSerializer* rejects an empty question field.

**Expected Behavior**: The serializer should be invalid and return a validation error for the question field.

##### test\_style\_question\_serializer\_empty\_answer:

**Description**: Tests if the *StyleQuestionSerializer* rejects an empty answer field.

**Expected Behavior**: The serializer should be invalid and return a validation error for the answer field.

#### Test\_view.py

#### QuestionViewTest

**Setup**: Creates a *QuestionList* instance and sets the endpoint URL to test retrieving questions.

##### test\_get\_questions\_successful:

**Description**: Tests if the *QuestionView* endpoint correctly returns questions for a valid category.

**Expected Behavior**: The endpoint should return a **200 OK** response, and the questions in the response should match the specified category.

##### test\_get\_questions\_invalid\_category:

**Description**: Tests if the *QuestionView* endpoint returns an error for an invalid category.

**Expected Behavior**: The endpoint should return a **400 Bad Request** response, indicating that the provided category is not valid.

#### SaveStyleViewTest

**Setup**: Creates a user, authenticates the client, and prepares valid and invalid payloads for testing style saving.

test\_save\_style\_successful:

**Description**: Tests if the *SaveStyleView* endpoint successfully creates a style with valid input data.

**Expected Behavior**: The endpoint should return a **201 Created** response with a success message.

##### test\_save\_style\_invalid\_data:

**Description**: Tests if the *SaveStyleView* endpoint handles invalid input data correctly (e.g., missing required fields).

**Expected Behavior**: The endpoint should return a **400 Bad Request** response with detailed error messages indicating the validation issues.

#### DeleteStyleViewTest

**Setup**: Creates a user and a Style instance, authenticates the client, and sets the endpoint URL to test deleting a style.

##### test\_delete\_style\_successful:

**Description**: Tests if the *DeleteStyleView* endpoint successfully deletes a style owned by the authenticated user.

**Expected Behavior**: The endpoint should return a **200 OK** response with a success message.

##### test\_delete\_style\_not\_found:

**Description**: Tests if the *DeleteStyleView* endpoint handles cases where the specified style ID does not exist or does not belong to the authenticated user.

**Expected Behavior**: The endpoint should return a **404 Not Found** response with an appropriate error message.

### Personalization

#### Test\_model.py

##### test\_personalization\_creation

**Description**: Ensures that a Personalization instance is created successfully with valid data and is correctly linked to a user.

**Setup**: A user is created with the username testuser. A Personalization instance is created with valid answers for all 10 questions.

**Expected Behavior**: The Personalization instance is successfully created and contains the correct data.

##### test\_personalization\_str\_method

**Description**: Verifies that the \_\_str\_\_ method of the *Personalization* model returns the correct string format.

**Setup**: A user is created with the username testuser. A *Personalization* instance is created for that user.

**Expected Behavior**: The string representation includes the username of the user associated with the *Personalization* instance.

#### Test\_serializer.py

##### test\_valid\_serializer

**Description**: Tests if the *PersonalizationSerializer* accepts and validates a payload with valid data.

**Setup**: A valid payload is prepared with proper answers for all 10 questions.

**Expected Behavior**: The serializer should validate the data without any errors.

##### test\_invalid\_serializer

**Description**: Tests if the *PersonalizationSerializer* correctly identifies errors in a payload with invalid data.

**Setup**: An invalid payload is prepared with incorrect values for question\_1 and question\_2.

**Expected Behavior**: The serializer should flag validation errors for the specified fields.

#### Test\_view.py

#### PersonalizationQuestionsViewTest

##### test\_get\_personalization\_questions

**Description**: Tests if the *PersonalizationQuestionsView* correctly returns a list of questions for authenticated users.

**Setup**: A user is created and authenticated. The personalization-questions endpoint is targeted.

**Expected Behavior**: The API returns a list of questions in the response.

#### SavePersonalizationViewTest

##### test\_save\_personalization\_success

**Description**: Tests if the *SavePersonalizationView* successfully saves personalization data for authenticated users with valid input.

**Setup**: A user is created and authenticated. A valid payload is prepared with answers for all 10 questions. Confirms that a new Personalization instance is created in the database.

**Expected Behavior**: The API successfully saves the data and returns a successful response.

##### test\_save\_personalization\_invalid

**Description**: Tests if the *SavePersonalizationView* correctly handles invalid input data.

**Setup**: A user is created and authenticated. An invalid payload is prepared with incorrect data for question\_1.

**Expected Behavior**: The API returns validation errors for the invalid data and does not create a Personalization instance.

## Frontend

### auth.js API Tests

This file contains tests for three API functions in the auth.js module: loginUserAPI, registerUserAPI, and logoutUserAPI. The tests check if the correct API calls are made with the appropriate parameters and verify that the correct responses are returned. Mocking is used for external dependencies like axiosInstance and localStorage.

1. Test for loginUserAPI - this test checks whether the loginUserAPI function correctly calls the login API endpoint with the provided user credentials.

2. Test for registerUserAPI - this test checks whether the registerUserAPI function correctly calls the registration API endpoint with the provided user data.

3. Test for logoutUserAPI - this test checks whether the logoutUserAPI function correctly calls the logout API endpoint with the provided refresh token.

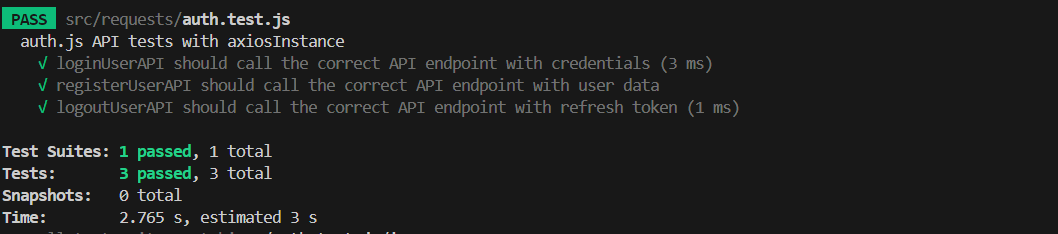


Figure 8 test auth.test.js

### axiosInstance.js Tests

This test suite is designed to test the behavior of an Axios instance, focusing on the request and response interceptors, and how they interact with localStorage and handle authentication tokens. The tests mock Axios, localStorage, and the atob function to simulate different scenarios, such as token validation and refreshing, and error handling in API responses.

Test "Should add Authorization header with valid access token from localStorage" adds Authorization header if tokens exist and are valid checks that if a valid access token is found in localStorage, the request interceptor will add the authorization header with the token. Simulation of a valid token stored in localStorage and mock the atob function to decode the token's payload. The interceptor adds the Authorization header to the request config with the valid access token.

Test "Should refresh access token and update Authorization header when access token is expired" refreshes token if access token is expired. This test checks that if the access token is expired, the request interceptor will use the refresh token to request a new access token and update the Authorization header. Simulated an expired access token and a valid refresh token in localStorage. The axiosInstance.post method is mocked to return a new access token. The Authorization header is updated with the new access token, and the authTokens in localStorage is updated with the new token.

Test "Should handle API errors by rejecting with error response data" handles API errors and returns a rejected promise. This test checks that the response interceptor correctly handles API errors and returns the error data in a rejected promise. We simulated an API error response and manually called the error handler in the response interceptor. The interceptor throws the error data in a rejected promise.

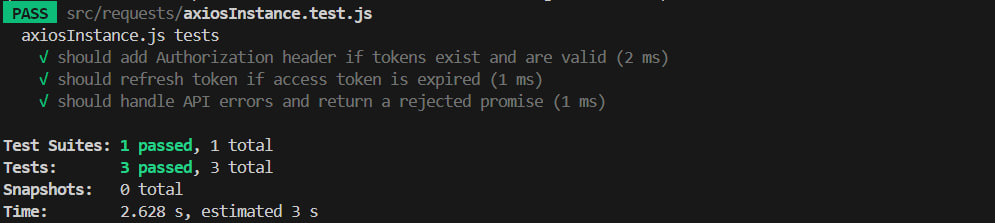


Figure 9 Test axiosInstance.test.js

### profile.js Tests

The unit tests for the Profile Module, which includes functions for fetching and updating a user's profile data using axios for API communication.

Test "fetch user profile successfully when API request succeeds" - verifies that the getUserProfile function correctly fetches the user profile when the API request is successful.

This test:

1. Mocks the get method to resolve with a predefined user profile response.
2. Calls the getUserProfile function.
3. Asserts that the get method was called with the correct URL and the result matches the mocked response.

Test "update user profile successfully when API request succeeds" verifies that the updateUserProfile function correctly updates the user profile when the API request is successful.

This test:

1. Mocks the put method to resolve with a successful response.
2. Calls the updateUserProfile function with valid data.
3. Asserts that the put method was called with the correct URL and the update data.
4. Verifies that the result matches the success response.

Test "handle error when fetching user profile fails" verifies that the getUserProfile function handles errors properly when the API request fails (e.g., user not found).

This test:

1. Mock the get method to reject with an error response.
2. Call the getUserProfile function.
3. Assert that the error message is handled correctly.

Test "handle error when updating user profile fails" verifies that the updateUserProfile function handles errors properly when the API request fails (e.g. invalid email format).

This test:

1. Mock the put method to reject with an error response.
2. Call the updateUserProfile function with invalid data.
3. Assert that the error message is handled correctly.

Test "refresh token and retry fetching user profile when token is expired" - verifies that the system correctly handles token expiration by refreshing the token and retrying the request.

This test:

1. Mock the first get request to reject with an expired token error.
2. Mock the post method to resolve with a new access token.
3. Mock the get method again to resolve with the user profile data after the token is refreshed.
4. Call the getUserProfile function and verify that the user profile is fetched successfully after retrying.

Test "handle empty or invalid data when updating user profile" verify that the updateUserProfile function handles cases where the input data is empty or invalid.

This test:

1. Mock the put method to resolve with an error response indicating invalid data.
2. Call the updateUserProfile function with empty data.
3. Assert that the error response is correctly handled.

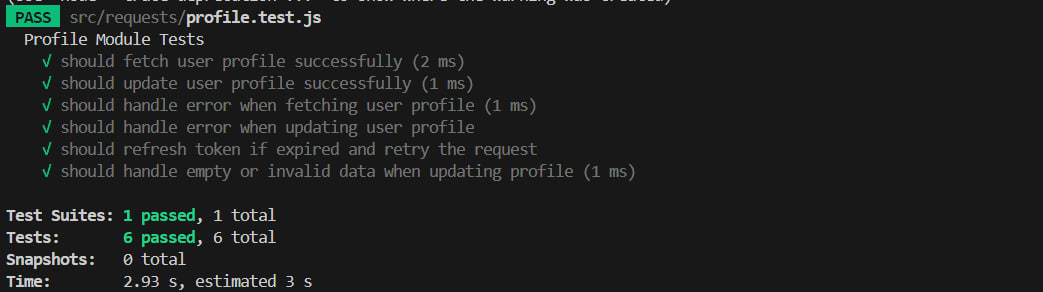


Figure 10 Test profile.test.js

# Conclusion

The ULetter facility is a breakthrough toward the simplification of formal letter preparation for students and professionals alike. By responding to the headaches created by proper formatting, tone, and structure, this application serves an emerging need both in academia and the professional world.

Key successes are: 1. A user-friendly interface with personalization options. 2. Integration of modern technologies like Django, React, and ChatGPT API. 3. Security, responsiveness, and reliability are ensured by rigorous testing.

ULetter is soon going to be the tool of choice for anyone in need of effective formal communications, with clear usability and functionality. This is a platform that, in further development, will be constantly enhanced by even more advanced features in correspondence with emerging user needs.