Store API Documentation

David Toscano

Apr 13, 2025

Welcome to the Store API documentation.

This documentation provides an overview of the Store API, its modules, and how to use it effectively. It is designed to help developers understand the structure and functionality of the API, as well as provide guidance on how to extend and customize it for their own needs. The Store API is a RESTful API built with FastAPI, designed to manage and serve data for a store application. It provides endpoints for user authentication, data management, and background tasks, among other features.

The API is built using Python and FastAPI, and it is designed to be easy to use and extend. The API is structured into several modules, each responsible for a specific aspect of the application. The modules are organized to provide a clear separation of concerns, making it easier to maintain and extend the codebase.

CHAPTER ONE

INTRODUCTION

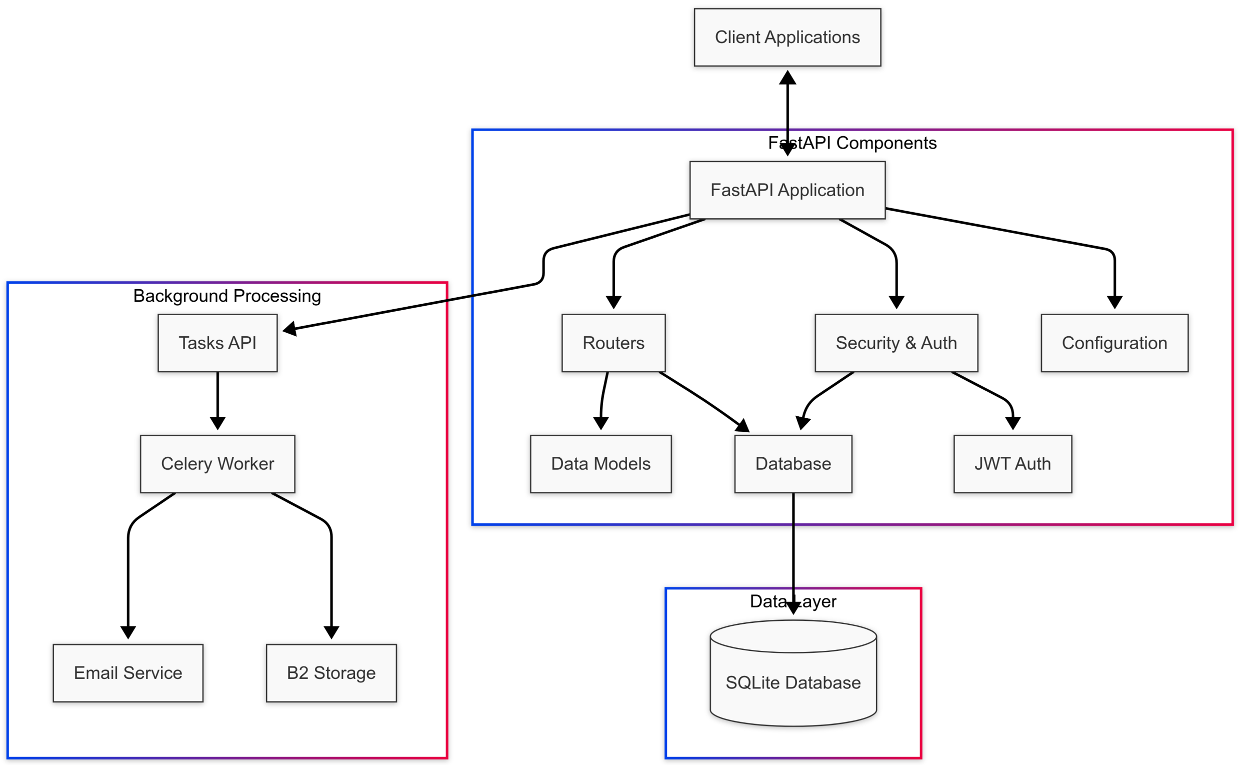
Store API is an application that allows users to create posts, upload images, register and authenticate users, and more.

1.1 Project Architecture

The Store API is built with FastAPI and follows a modular architecture:

- **Routers**: Handle HTTP requests and responses
- **Models**: Define data structures
- **Database**: Manages database connections and operations
- **Security**: Handles authentication and authorization
- **Tasks**: Manages background tasks with Celery

1.2 Architecture Diagram



CHAPTER

TWO

MAIN MODULE

The main application module that initializes FastAPI. Main FastAPI application module for the Store API. This module configures and initializes the FastAPI application with its routes and middleware.

`async storeapi.main.custom_http_exception_handler(`*request*, *exc*`)`

Custom exception handler for HTTP exceptions. Logs the exception details and then delegates to the default handler.

Parameters

- `request` – The request that caused the exception
- `exc` – The HTTP exception that was raised

Returns

The response from the default HTTP exception handler

`storeapi.main.lifespan(`*app_instance: FastAPI*`)`

Manages the startup and shutdown events of the FastAPI application. Connects to the database on startup and disconnects on shutdown.

Parameters

`app_instance` – The FastAPI application instance (required by FastAPI)

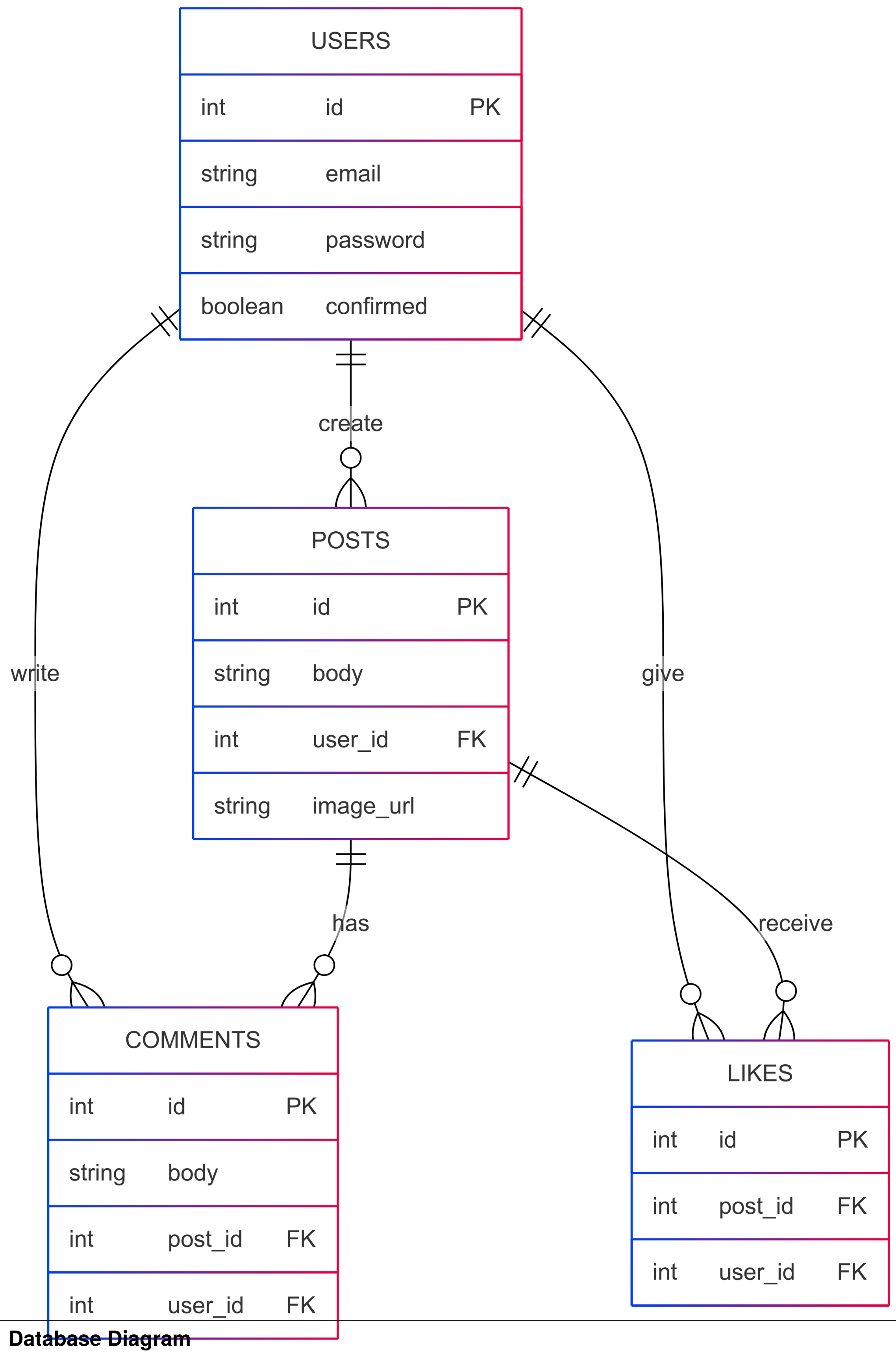
CHAPTER

THREE

DATABASE

This module defines database models, tables, and connection configuration. It includes tables for users, posts, comments, and likes with their relationships.

3.1 Database Diagram



3.2 API Documentation

Database module for the Store API application. This module defines database models, tables, and connection configuration. It includes tables for users, posts, comments, and likes with their relationships.

CHAPTER

FOUR

CONFIGURATION

4.1 Configuration Values

This module defines environment-specific settings and configuration classes. Configuration module for the Store API application. This module defines environment-specific settings and configuration classes.

class storeapi.config.**BaseConfig**(*_case_sensitive: bool | None = None, _nested_model_default_partial_update: bool | None = None, _env_prefix: str | None = None, _env_file: DotenvType | None = WindowsPath('.'), _env_file_encoding: str | None = None, _env_ignore_empty: bool | None = None, _env_nested_delimiter: str | None = None, _env_parse_none_str: str | None = None, _env_parse_enums: bool | None = None, _cli_prog_name: str | None = None, _cli_parse_args: bool | list[str] | tuple[str, ...] | None = None, _cli_settings_source: CliSettingsSource[Any] | None = None, _cli_parse_none_str: str | None = None, _cli_hide_none_type: bool | None = None, _cli_avoid_json: bool | None = None, _cli_enforce_required: bool | None = None, _cli_use_class_docs_for_groups: bool | None = None, _cli_exit_on_error: bool | None = None, _cli_prefix: str | None = None, _cli_flag_prefix_char: str | None = None, _cli_implicit_flags: bool | None = None, _cli_ignore_unknown_args: bool | None = None, _secrets_dir: PathType | None = None, *, ENV_STATE: str | None = None*)

Bases: `BaseSettings`

Base configuration class for the application. It uses Pydantic's BaseSettings to load environment variables.

ENV_STATE: str | None

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'cli_avoid_json': False, 'cli_enforce_required': False, 'cli_exit_on_error': True, 'cli_flag_prefix_char': '-', 'cli_hide_none_type': False, 'cli_ignore_unknown_args': False, 'cli_implicit_flags': False, 'cli_parse_args': None, 'cli_parse_none_str': None, 'cli_prefix': '', 'cli_prog_name': None, 'cli_use_class_docs_for_groups': False, 'env_file': '.env', 'env_file_encoding': 'utf-8', 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_enums': None, 'env_parse_none_str': None, 'env_prefix': '', 'extra': 'ignore', 'json_file': None, 'json_file_encoding': None, 'nested_model_default_partial_update': False, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class storeapi.config.**DevConfig**(*_case_sensitive: bool | None = None, _nested_model_default_partial_update: bool | None = None, _env_prefix: str | None = None, _env_file: DotenvType | None = WindowsPath('.'), _env_file_encoding: str | None = None, _env_ignore_empty: bool | None = None, _env_nested_delimiter: str | None = None, _env_parse_none_str: str | None = None, _env_parse_enums: bool | None = None, _cli_prog_name: str | None = None, _cli_parse_args: bool | list[str] | tuple[str, ...] | None = None, _cli_settings_source: CliSettingsSource[Any] | None = None, _cli_parse_none_str: str | None = None, _cli_hide_none_type: bool | None = None, _cli_avoid_json: bool | None = None, _cli_enforce_required: bool | None = None, _cli_use_class_docs_for_groups: bool | None = None, _cli_exit_on_error: bool | None = None, _cli_prefix: str | None = None, _cli_flag_prefix_char: str | None = None, _cli_implicit_flags: bool | None = None, _cli_ignore_unknown_args: bool | None = None, _secrets_dir: PathType | None = None, *, ENV_STATE: str | None = None, DATABASE_URL: str | None = None, DB_FORCE_ROLL_BACK: bool = False, LOGTAIL_API_KEY: str | None = None, LOGTAIL_INGESTING_HOST: str | None = None, MAILGUN_API_KEY: str | None = None, MAILGUN_DOMAIN: str | None = None, CELERY_BROKER_URL: str | None = None, CELERY_RESULT_BACKEND: str | None = None, B2_KEY_ID: str | None = None, B2_APPLICATION_KEY: str | None = None, B2_BUCKET_NAME: str | None = None*)

Bases: `GlobalConfig`

Development configuration class.

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'cli_avoid_json': False, 'cli_enforce_required': False, 'cli_exit_on_error': True, 'cli_flag_prefix_char': '-', 'cli_hide_none_type': False, 'cli_ignore_unknown_args': False, 'cli_implicit_flags': False, 'cli_parse_args': None, 'cli_parse_none_str': None, 'cli_prefix': '', 'cli_prog_name': None, 'cli_use_class_docs_for_groups': False, 'env_file': '.env', 'env_file_encoding': 'utf-8', 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_enums': None, 'env_parse_none_str': None, 'env_prefix': 'DEV_', 'extra': 'ignore', 'json_file': None, 'json_file_encoding': None, 'nested_model_default_partial_update': False, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class storeapi.config.**GlobalConfig**(*_case_sensitive: bool | None = None, _nested_model_default_partial_update: bool | None = None, _env_prefix: str | None = None, _env_file: DotenvType | None = WindowsPath('.'), _env_file_encoding: str | None = None, _env_ignore_empty: bool | None = None, _env_nested_delimiter: str | None = None, _env_parse_none_str: str | None = None, _env_parse_enums: bool | None = None, _cli_prog_name: str | None = None, _cli_parse_args: bool | list[str] | tuple[str, ...] | None = None, _cli_settings_source: CliSettingsSource[Any] | None = None, _cli_parse_none_str: str | None = None, _cli_hide_none_type: bool | None = None, _cli_avoid_json: bool | None = None, _cli_enforce_required: bool | None = None, _cli_use_class_docs_for_groups: bool | None = None, _cli_exit_on_error: bool | None = None, _cli_prefix: str | None = None, _cli_flag_prefix_char: str | None = None, _cli_implicit_flags: bool | None = None, _cli_ignore_unknown_args: bool | None = None, _secrets_dir: PathType | None = None, *, ENV_STATE: str | None = None, DATABASE_URL: str | None = None, DB_FORCE_ROLL_BACK: bool = False, LOGTAIL_API_KEY: str | None = None, LOGTAIL_INGESTING_HOST: str | None = None, MAILGUN_API_KEY: str | None = None, MAILGUN_DOMAIN: str | None = None, CELERY_BROKER_URL: str | None = None, CELERY_RESULT_BACKEND: str | None = None, B2_KEY_ID: str | None = None, B2_APPLICATION_KEY: str | None = None, B2_BUCKET_NAME: str | None = None*)

Bases: `BaseConfig`

Global configuration settings for all environments. Contains common configuration parameters used across the application.

B2_APPLICATION_KEY: str | None

B2_BUCKET_NAME: str | None

B2_KEY_ID: str | None

CELERY_BROKER_URL: str | None

CELERY_RESULT_BACKEND: str | None

DATABASE_URL: str | None

DB_FORCE_ROLL_BACK: bool

LOGTAIL_API_KEY: str | None

LOGTAIL_INGESTING_HOST: str | None

MAILGUN_API_KEY: str | None

MAILGUN_DOMAIN: str | None

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'cli_avoid_json': False, 'cli_enforce_required': False, 'cli_exit_on_error': True, 'cli_flag_prefix_char': '-', 'cli_hide_none_type': False, 'cli_ignore_unknown_args': False, 'cli_implicit_flags': False, 'cli_parse_args': None, 'cli_parse_none_str': None, 'cli_prefix': '', 'cli_prog_name': None, 'cli_use_class_docs_for_groups': False, 'env_file': '.env', 'env_file_encoding': 'utf-8', 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_enums': None, 'env_parse_none_str': None, 'env_prefix': '', 'extra': 'ignore', 'json_file': None, 'json_file_encoding': None, 'nested_model_default_partial_update': False, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class storeapi.config.**ProdConfig**(*_case_sensitive: bool | None = None, _nested_model_default_partial_update: bool | None = None, _env_prefix: str | None = None, _env_file: DotenvType | None = WindowsPath('.'), _env_file_encoding: str | None = None, _env_ignore_empty: bool | None = None, _env_nested_delimiter: str | None = None, _env_parse_none_str: str | None = None, _env_parse_enums: bool | None = None, _cli_prog_name: str | None = None, _cli_parse_args: bool | list[str] | tuple[str, ...] | None = None, _cli_settings_source: CliSettingsSource[Any] | None = None, _cli_parse_none_str: str | None = None, _cli_hide_none_type: bool | None = None, _cli_avoid_json: bool | None = None, _cli_enforce_required: bool | None = None, _cli_use_class_docs_for_groups: bool | None = None, _cli_exit_on_error: bool | None = None, _cli_prefix: str | None = None, _cli_flag_prefix_char: str | None = None, _cli_implicit_flags: bool | None = None, _cli_ignore_unknown_args: bool | None = None, _secrets_dir: PathType | None = None, *, ENV_STATE: str | None = None, DATABASE_URL: str | None = None, DB_FORCE_ROLL_BACK: bool = False, LOGTAIL_API_KEY: str | None = None, LOGTAIL_INGESTING_HOST: str | None = None, MAILGUN_API_KEY: str | None = None, MAILGUN_DOMAIN: str | None = None, CELERY_BROKER_URL: str | None = None, CELERY_RESULT_BACKEND: str | None = None, B2_KEY_ID: str | None = None, B2_APPLICATION_KEY: str | None = None, B2_BUCKET_NAME: str | None = None*)

Bases: `GlobalConfig`

Production configuration class.

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'cli_avoid_json': False, 'cli_enforce_required': False, 'cli_exit_on_error': True, 'cli_flag_prefix_char': '-', 'cli_hide_none_type': False, 'cli_ignore_unknown_args': False, 'cli_implicit_flags': False, 'cli_parse_args': None, 'cli_parse_none_str': None, 'cli_prefix': '', 'cli_prog_name': None, 'cli_use_class_docs_for_groups': False, 'env_file': '.env', 'env_file_encoding': 'utf-8', 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_enums': None, 'env_parse_none_str': None, 'env_prefix': 'PROD_', 'extra': 'ignore', 'json_file': None, 'json_file_encoding': None, 'nested_model_default_partial_update': False, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class storeapi.config.**TestConfig**(*_case_sensitive: bool | None = None, _nested_model_default_partial_update: bool | None = None, _env_prefix: str | None = None, _env_file: DotenvType | None = WindowsPath('.'), _env_file_encoding: str | None = None, _env_ignore_empty: bool | None = None, _env_nested_delimiter: str | None = None, _env_parse_none_str: str | None = None, _env_parse_enums: bool | None = None, _cli_prog_name: str | None = None, _cli_parse_args: bool | list[str] | tuple[str, ...] | None = None, _cli_settings_source: CliSettingsSource[Any] | None = None, _cli_parse_none_str: str | None = None, _cli_hide_none_type: bool | None = None, _cli_avoid_json: bool | None = None, _cli_enforce_required: bool | None = None, _cli_use_class_docs_for_groups: bool | None = None, _cli_exit_on_error: bool | None = None, _cli_prefix: str | None = None, _cli_flag_prefix_char: str | None = None, _cli_implicit_flags: bool | None = None, _cli_ignore_unknown_args: bool | None = None, _secrets_dir: PathType | None = None, *, ENV_STATE: str | None = None, DATABASE_URL: str = 'sqlite:///test.db', DB_FORCE_ROLL_BACK: bool = True, LOGTAIL_API_KEY: str | None = None, LOGTAIL_INGESTING_HOST: str | None = None, MAILGUN_API_KEY: str | None = None, MAILGUN_DOMAIN: str | None = None, CELERY_BROKER_URL: str | None = None, CELERY_RESULT_BACKEND: str | None = None, B2_KEY_ID: str | None = None, B2_APPLICATION_KEY: str | None = None, B2_BUCKET_NAME: str | None = None*)

Bases: *GlobalConfig*

Testing configuration class.

DATABASE_URL: str

DB_FORCE_ROLL_BACK: bool

model_config: ClassVar[SettingsConfigDict] = {'arbitrary_types_allowed': True, 'case_sensitive': False, 'cli_avoid_json': False, 'cli_enforce_required': False, 'cli_exit_on_error': True, 'cli_flag_prefix_char': '-', 'cli_hide_none_type': False, 'cli_ignore_unknown_args': False, 'cli_implicit_flags': False, 'cli_parse_args': None, 'cli_parse_none_str': None, 'cli_prefix': '', 'cli_prog_name': None, 'cli_use_class_docs_for_groups': False, 'env_file': '.env', 'env_file_encoding': 'utf-8', 'env_ignore_empty': False, 'env_nested_delimiter': None, 'env_parse_enums': None, 'env_parse_none_str': None, 'env_prefix': 'TEST_', 'extra': 'ignore', 'json_file': None, 'json_file_encoding': None, 'nested_model_default_partial_update': False, 'protected_namespaces': ('model_', 'settings_'), 'secrets_dir': None, 'toml_file': None, 'validate_default': True, 'yaml_file': None, 'yaml_file_encoding': None}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

storeapi.config.**get_config**(*env_state: str*)

Function to get the appropriate configuration class based on the environment state.

4.2 Logging Configuration

This module configures loggers, handlers, formatters and filters for comprehensive application logging including console, file and external service outputs. Logging configuration module for the Store API application. This module configures loggers, handlers, formatters and filters for comprehensive application logging including console, file and external service outputs.

class `storeapi.logging_conf.`**`EmailObfuscationFilter`**(*name: str = ''*, *obfuscated_length: int = 2*)

Bases: `Filter`

Logging filter that obfuscates email addresses in log records.

This filter examines log records for email fields and applies obfuscation to protect user privacy while maintaining diagnostic value. The obfuscation level is configurable depending on the environment.

`filter`(*record: LogRecord*) → bool

Determine if the specified record is to be logged.

Returns True if the record should be logged, or False otherwise. If deemed appropriate, the record may be modified in-place.

`storeapi.logging_conf.`**`configure_logging`**() → None

Configure application-wide logging settings.

Sets up different logging handlers (console, file, external services), formatters with correlation IDs, and specific logger configurations for various components. The configuration adapts based on the current environment (development/production).

`storeapi.logging_conf.`**`obfuscated`**(*email: str*, *obfuscated_length: int*)

Obfuscate email address for logging purposes.

CHAPTER

FIVE

SECURITY

This module handles security concerns including authentication, authorization, and token management. Security module for the Store API application. This module handles authentication, authorization, and token management including JWT generation, password hashing, and user verification.

`storeapi.security.`**`access_token_expire_minutes`**`()` → int

Get the expiration time in minutes for access tokens.

Returns
Number of minutes until access tokens expire

Return type
int

`async` `storeapi.security.`**`authenticate_user`**`(`*email: str*, *password: str*`)`

Authenticate a user with email and password.

Parameters

- **`email`** – User's email address
- **`password`** – User's plain text password

Returns
User record if authentication succeeds

Return type
dict

Raises
`HTTPException` – If authentication fails for any reason

`storeapi.security.`**`confirm_token_expire_minutes`**`()` → int

Get the expiration time in minutes for confirmation tokens.

Returns
Number of minutes until confirmation tokens expire (24 hours)

Return type
int

`storeapi.security.`**`create_access_token`**`(`*email: str*`)` → str

Create a JWT access token for the specified user.

Parameters
`email` – User's email address to encode in the token

Returns
Encoded JWT token with user email and expiration time

Return type
str

`storeapi.security.`**`create_confirmation_token`**(*email: str*) → str

Create a JWT confirmation token for email verification.

Parameters
`email` – User’s email address to encode in the token

Returns
Encoded JWT token with user email and longer expiration time

Return type
str

`storeapi.security.`**`create_credentials_exception`**(*detail: str*) → HTTPException

Create a standardized HTTP 401 exception for authentication failures.

Parameters
`detail` – Specific error message explaining why authentication failed

Returns
Properly formatted 401 Unauthorized exception

Return type
HTTPException

`async storeapi.security.`**`get_current_user`**(*token: str*)

Get the current authenticated user from a JWT token.

Parameters
`token` – JWT access token from request

Returns
User record of the authenticated user

Return type
dict

Raises
`HTTPException` – If token is invalid or user doesn’t exist

`storeapi.security.`**`get_password_hash`**(*password: str*) → str

Generate a secure hash of the provided password.

Parameters
`password` – Plain text password to hash

Returns
Securely hashed password

Return type
str

`storeapi.security.`**`get_subject_for_token_type`**(*token: str*, *token_type: Literal['access', 'confirmation']*) → str

Validate a JWT token and extract the subject if token type matches.

Parameters

- **`token`** – JWT token to validate and decode
- **`token_type`** – Expected token type (‘access’ or ‘confirmation’)

Returns
Email address extracted from the token

Return type
str

Raises
`HTTPException` – If token is invalid, expired, or wrong type

`async storeapi.security.get_user`(*email: str*)

Retrieve a user from the database by email.

Parameters
`email` – Email address of the user to find

Returns
User record if found, None otherwise

Return type
dict

`storeapi.security.verify_password`(*plain_password: str*, *hashed_password: str*) → bool

Verify that a plain password matches a hashed password.

Parameters

- `plain_password` – Plain text password to verify
- `hashed_password` – Previously hashed password to check against

Returns
True if passwords match, False otherwise

Return type
bool

CHAPTER

SIX

BACKGROUND TASKS

This module handles asynchronous background tasks using Celery. Tasks module for the Store API application. This module defines Celery tasks for asynchronous processing, including email sending and image generation features.

exception `storeapi.tasks.`**`APIResponseError`**

Bases: `Exception`

Exception raised when an API request fails. Provides information about HTTP status codes and error details.

6.1 Celery Configuration

Celery configuration module for the Store API application. This module initializes and configures the Celery instance for background task processing.

CHAPTER

SEVEN

API ROUTERS

7.1 User Router

User router module for the Store API application. Handles user registration, authentication, and email confirmation. Provides endpoints for creating user accounts and managing user sessions.

`async storeapi.routers.user.confirm_email(token)`

Confirm a user's email address using the provided token.

Parameters
`token` – Email confirmation token sent to the user

Returns
JSON response with confirmation status message

Raises
`HTTPException` – If token is invalid (handled by get_subject_for_token_type)

`async storeapi.routers.user.login(form_data: OAuth2PasswordRequestForm)`

Authenticate a user and provide an access token.

Parameters
`form_data` – OAuth2 form containing username (email) and password

Returns
JSON response with access token and token type

Raises
`HTTPException` – If authentication fails (handled by authenticate_user)

`async storeapi.routers.user.register(user: UserIn, request: Request)`

Register a new user in the system.

Parameters

- `user` – User data including email and password
- `request` – FastAPI request object to generate confirmation URL

Returns
JSON response with registration confirmation message

Raises
`HTTPException` – If a user with the provided email already exists

7.2 Post Router

Post router module for the Store API application. Handles all HTTP requests related to posts, comments, and likes. Provides endpoints for creating, retrieving, and interacting with user posts.

`class storeapi.routers.post.`**`PostSorting`**`(`*value*, *names=<not given>*, **values*, *module=None*, *qualname=None*, *type=None*, *start=1*, *boundary=None*`)`

Bases: `str`, `Enum`

Enumeration for post sorting options. Defines the available sorting mechanisms for post listings.

`MOST_LIKES = 'most_likes'`

`NEW = 'new'`

`OLD = 'old'`

`async storeapi.routers.post.`**`create_comment`**`(`*comment: CommentIn*, *current_user: User*`)`

Create a new comment on a post.

Parameters

- **`comment`** – The comment content and post ID
- **`current_user`** – The authenticated user making the comment

Returns

The created comment data including the new ID

Raises

`HTTPException` – If the post doesn't exist

`async storeapi.routers.post.`**`create_post`**`(`*post: UserPostIn*, *current_user: User*, *request: Request*, *prompt: str = None*`)`

Create a new post with optional image generation.

Parameters

- **`post`** – The post content to create
- **`current_user`** – The authenticated user making the request
- **`request`** – The FastAPI request object
- **`prompt`** – Optional text prompt for image generation

Returns

The created post data including the new ID

`async storeapi.routers.post.`**`find_post`**`(`*post_id: int*`)`

Find a post by its ID in the database.

Parameters

`post_id` – The ID of the post to find

Returns

The post record if found, None otherwise

`async storeapi.routers.post.`**`get_all_posts`**`(`*sorting:* PostSorting = *PostSorting.NEW*`)`

Get all posts with optional sorting.

Parameters

`sorting` – The sorting method to use (new, old, or most_likes)

Returns
List of posts with like counts, sorted according to the specified criteria

async storeapi.routers.post.**get_comments_on_post**(*post_id: int*)

Get all comments for a specific post.

Parameters
post_id – The ID of the post to retrieve comments for

Returns
List of comments on the specified post

async storeapi.routers.post.**get_post_with_comments**(*post_id: int*)

Get a post and all its comments.

Parameters
post_id – The ID of the post to retrieve

Returns
The post with its like count and all comments

Raises
HTTPException – If the post doesn't exist

async storeapi.routers.post.**like_post**(*post_like: PostLikeIn*, *current_user: User*)

Like a post.

Parameters

- **post_like** – The post ID to like
- **current_user** – The authenticated user performing the like action

Returns
The created like data including the new ID

Raises
HTTPException – If the post doesn't exist

7.3 Upload Router

Upload router module for the Store API application. Handles file uploads from clients, processes files in chunks, and uploads them to Backblaze B2 cloud storage.

async storeapi.routers.upload.**upload_file**(*file: UploadFile*)

Process an uploaded file in chunks and store it in B2 cloud storage.

Parameters
file – The uploaded file from the client

Returns
A dictionary with upload confirmation and file URL

Raises
HTTPException – If there's an error during the file upload process

CHAPTER

EIGHT

MODELS

8.1 User Models

User model definitions for the Store API application. Contains Pydantic models that define the structure of user data for both requests and responses.

class `storeapi.models.user.`**`User`**(*, *id: int*, *email: str*)

Bases: `BaseModel`

User response model representing a user in the system. Contains the user's ID and email address, but omits sensitive information like passwords for security.

`email: str`

`id: int`

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class `storeapi.models.user.`**`UserIn`**(*, *email: str*, *password: str*)

Bases: `BaseModel`

User input model for user registration and authentication. Contains the user's email address and password which are required for creating a new user account or authenticating an existing user.

`email: str`

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`password: str`

8.2 Post Models

Post model definitions for the Store API application. Contains Pydantic models for user posts, comments, and likes that define the structure of request and response data.

class `storeapi.models.post.`**`Comment`**`(*, `*body: str*, *post_id: int*, *id: int*, *user_id: int*`)`

Bases: `CommentIn`

Complete comment model that extends the input model. Includes server-generated fields like ID and user_id.

`id: int`

`model_config: ClassVar[ConfigDict] = {'from_attributes': True}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`user_id: int`

class `storeapi.models.post.`**`CommentIn`**`(*, `*body: str*, *post_id: int*`)`

Bases: `BaseModel`

Input model for creating a new comment. Contains the comment content and associated post ID.

`body: str`

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`post_id: int`

class `storeapi.models.post.`**`PostLike`**`(*, `*post_id: int*, *id: int*, *user_id: int*`)`

Bases: `PostLikeIn`

Complete post like model that extends the input model. Includes server-generated fields like ID and user_id.

`id: int`

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`user_id: int`

class `storeapi.models.post.`**`PostLikeIn`**`(*, `*post_id: int*`)`

Bases: `BaseModel`

Input model for creating a new post like. Contains only the post ID to be liked.

`model_config: ClassVar[ConfigDict] = {}`

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

`post_id: int`

class `storeapi.models.post.`**`UserPost`**`(*, `*body: str*, *id: int*, *user_id: int*, *image_url: str | None = None*`)`

Bases: `UserPostIn`

Complete user post model that extends the input model. Includes server-generated fields like ID, user_id, and optional image URL.

id: int

image_url: str | None

model_config: ClassVar[ConfigDict] = {'from_attributes': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

user_id: int

class storeapi.models.post.**UserPostIn**(*, *body: str*)

Bases: BaseModel

Input model for creating a new user post. Contains only the post content body.

body: str

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

class storeapi.models.post.**UserPostWithComments**(*, *post: UserPostWithLikes*, *comments: list[Comment]*)

Bases: BaseModel

Composite model that contains a post with its like count and all associated comments.

comments: list[Comment]

model_config: ClassVar[ConfigDict] = {}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

post: UserPostWithLikes

class storeapi.models.post.**UserPostWithLikes**(*, *body: str*, *id: int*, *user_id: int*, *image_url: str | None = None*, *likes: int*)

Bases: UserPost

User post model that includes like count information. Extends the base UserPost model with a likes field.

likes: int

model_config: ClassVar[ConfigDict] = {'from_attributes': True}

Configuration for the model, should be a dictionary conforming to [*ConfigDict*][pydantic.config.ConfigDict].

CHAPTER

NINE

LIBRARIES

9.1 B2 Cloud Storage

Backblaze B2 Cloud Storage integration module. Provides utilities for authenticating, connecting to buckets, and uploading files to the B2 cloud storage service.

`storeapi.libs.b2.`**`b2_api`**`()`

Create and authorize a B2 API client.

Returns
An authenticated B2 API client instance.

Return type
B2Api

`storeapi.libs.b2.`**`b2_get_bucket`**`(`*api: B2Api*`)`

Get a B2 bucket by name from the configuration.

Parameters
`api` – B2 API client instance

Returns
B2 bucket object

Return type
Bucket

`storeapi.libs.b2.`**`b2_upload_file`**`(`*local_file: str*, *file_name: str*`)` → str

Upload a local file to B2 cloud storage.

Parameters

- **`local_file`** – Path to the local file to upload
- **`file_name`** – Name to assign to the file in B2

Returns
Download URL for the uploaded file

Return type
str

CHAPTER

TEN

API TESTS

This module contains tests for the Store API, ensuring that all endpoints and functionalities work as expected.

10.1 Test Configuration

Pytest configuration file for the Store API test suite. Defines fixtures for test client setup, database connections, user management, and mocks for testing the application without external dependencies.

`storeapi.tests.conftest.`**`anyio_backend`**`()`

Configure pytest-anyio to use asyncio backend for async tests.

Returns
The name of the backend to use

Return type
str

`storeapi.tests.conftest.`**`async_client_fixture`**`(`*client_fixture*`)` → AsyncGenerator

Create an async HTTP client for testing async endpoints.

Parameters
`client_fixture` – The standard test client

Yields
AsyncClient – An async capable HTTP client

`storeapi.tests.conftest.`**`client_fixture`**`()` → Generator

Create a test client for the FastAPI application.

Yields
TestClient – A FastAPI test client

`storeapi.tests.conftest.`**`confirmed_user_fixture`**`(`*registered_user_fixture: dict*`)` → dict

Create a confirmed test user by updating a registered user's status.

Parameters
`registered_user_fixture` – The registered user details

Returns
User details with confirmed status

Return type
dict

`storeapi.tests.conftest.`**`db`**`()` → AsyncGenerator

Set up and tear down the database connection for each test. This fixture runs automatically for all tests.

Yields

None – Just establishes and closes database connection

`storeapi.tests.conftest.`**`logged_in_token_fixture`**`(`*async_client_fixture: AsyncClient*, *confirmed_user_fixture: dict*`)` → str

Obtain an authentication token for a confirmed user.

Parameters

- **`async_client_fixture`** – The async HTTP client
- **`confirmed_user_fixture`** – The confirmed user details

Returns

Authentication token for the user

Return type

str

`storeapi.tests.conftest.`**`mock_celery_tasks_fixture`**`(`*mocker*`)`

Mock celery tasks to avoid actual execution during tests.

Parameters

`mocker` – pytest-mock fixture

Returns

The mocked task

Return type

MagicMock

`storeapi.tests.conftest.`**`registered_user_fixture`**`(`*async_client_fixture: AsyncClient*`)` → dict

Create a registered but unconfirmed test user.

Parameters

`async_client_fixture` – The async HTTP client

Returns

User details including email, password and ID

Return type

dict

10.2 Security Tests

Security module test suite. Tests the functionality of token creation, authentication, password hashing, and user validation for the Store API application.

`storeapi.tests.test_security.`**`test_access_token_expire_minutes`**`()`

Test that access tokens are set to expire after 30 minutes.

`async` `storeapi.tests.test_security.`**`test_authenticate_user`**`(`*confirmed_user_fixture: dict*`)`

Test that a confirmed user can be authenticated with correct credentials.

Parameters

`confirmed_user_fixture` – Test user data with confirmed status async storeapi.tests.test_security.**test_authenticate_user_not_found**()

Test that authentication fails for non-existent users.

async storeapi.tests.test_security.**test_authenticate_user_wrong_password**(*confirmed_user_fixture: dict*)

Test that authentication fails with incorrect password.

Parameters
confirmed_user_fixture – Test user data with confirmed status

storeapi.tests.test_security.**test_confirm_token_expire_minutes**()

Test that confirmation tokens are set to expire after 1440 minutes (24 hours).

storeapi.tests.test_security.**test_create_token**(*create_token_func*, *token_type*)

Test that tokens are created with correct subject and type.

Parameters

- **create_token_func** – Function that creates the token
- **token_type** – Expected type of the token

async storeapi.tests.test_security.**test_get_current_user**(*confirmed_user_fixture: dict*)

Test that current user can be retrieved from a valid token.

Parameters
confirmed_user_fixture – Test user data with confirmed status

async storeapi.tests.test_security.**test_get_current_user_invalid_token**()

Test that get_current_user fails with invalid token.

async storeapi.tests.test_security.**test_get_current_user_wrong_type_token**(*confirmed_user_fixture: dict*)

Test that get_current_user fails with wrong token type.

Parameters
confirmed_user_fixture – Test user data with confirmed status

storeapi.tests.test_security.**test_get_subject_for_token_expired**(*mocker*)

Test that expired tokens are rejected with appropriate error message.

storeapi.tests.test_security.**test_get_subject_for_token_invalid**()

Test that invalid tokens are rejected with appropriate error message.

storeapi.tests.test_security.**test_get_subject_for_token_type_missing_sub**()

Test that tokens missing the subject field are rejected.

storeapi.tests.test_security.**test_get_subject_for_token_type_valid_confirmation**(*create_token_func*, *token_type*)

Test that token subject can be extracted correctly.

Parameters

- **create_token_func** – Function that creates the token
- **token_type** – Type of the token to validate

storeapi.tests.test_security.**test_get_subject_for_token_type_wrong_type**()

Test that tokens with incorrect type are rejected.

`async storeapi.tests.test_security.`**`test_get_user`**(*registered_user_fixture: dict*)

Test that a registered user can be retrieved from the database.

Parameters
`registered_user_fixture` – Test user data

`async storeapi.tests.test_security.`**`test_get_user_not_found`**()

Test that get_user returns None for non-existent users.

`storeapi.tests.test_security.`**`test_password_hashes`**()

Test that password hashing and verification work correctly.

10.3 Background Tasks Tests

Task module test suite. Tests email sending, image generation, and task processing functionality for the Store API application's asynchronous tasks.

`storeapi.tests.test_tasks.`**`mock_http_error_response`**()

Fixture that provides a simulated HTTP 500 error response.

`storeapi.tests.test_tasks.`**`test_generate_image_and_add_to_post_api_error`**(*mock_create_engine*, *mock_send_email*, *mock_generate_image*)

Test that verifies API error handling

`storeapi.tests.test_tasks.`**`test_generate_image_and_add_to_post_success`**(*mock_create_engine*, *mock_send_email*, *mock_generate_image*)

Test that verifies generate_image_and_add_to_post works properly

`storeapi.tests.test_tasks.`**`test_generate_image_and_add_to_post_with_existing_image`**(*mock_create_engine*, *mock_send_email*, *mock_generate_image*)

Test that verifies if the post already has an image, no new one is generated

`storeapi.tests.test_tasks.`**`test_generate_image_api_success`**()

Test that verifies _generate_image_api works properly with Pollinations.ai

`storeapi.tests.test_tasks.`**`test_send_simple_email`**()

Test that send_simple_email works correctly

`storeapi.tests.test_tasks.`**`test_send_simple_email_api_error`**()

Test that send_simple_email handles API errors correctly

`storeapi.tests.test_tasks.`**`test_send_user_registration_email`**()

Test that send_user_registration_email works correctly

10.4 Router Tests

Post router test module. Tests CRUD operations for posts, comments, and likes in the Store API. Verifies post creation, listing, sorting, commenting, and liking functionality.

`async storeapi.tests.routers.test_posts.`**`create_comment`**(*body: str*, *post_id: int*, *async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*) → dict

Helper function to create a comment on a post.

Parameters

- **`body`** – Text content of the comment
- **`post_id`** – ID of the post to comment on
- **`async_client_fixture`** – HTTP client for making requests
- **`logged_in_token_fixture`** – Authentication token

Returns
Created comment data

Return type
dict

`async storeapi.tests.routers.test_posts.`**`create_post`**(*body: str*, *async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*) → dict

Helper function to create a post.

Parameters

- **`body`** – Text content of the post
- **`async_client_fixture`** – HTTP client for making requests
- **`logged_in_token_fixture`** – Authentication token

Returns
Created post data

Return type
dict

`storeapi.tests.routers.test_posts.`**`created_comment_fixture`**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*, *logged_in_token_fixture: str*)

Fixture that creates a test comment on a post.

Parameters

- **`async_client_fixture`** – HTTP client for making requests
- **`created_post_fixture`** – Post to comment on
- **`logged_in_token_fixture`** – Authentication token

Returns
Created comment data

Return type
dict

`storeapi.tests.routers.test_posts.`**`created_post_fixture`**(*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*)

Fixture that creates a test post.

Parameters

- **`async_client_fixture`** – HTTP client for making requests
- **`logged_in_token_fixture`** – Authentication token

Returns
Created post data

Return type
dict

`async storeapi.tests.routers.test_posts.`**`like_post`**(*post_id: int*, *async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*) → dict

Helper function to like a post.

Parameters

- **`post_id`** – ID of the post to like
- **`async_client_fixture`** – HTTP client for making requests
- **`logged_in_token_fixture`** – Authentication token

Returns
Like data

Return type
dict

`async storeapi.tests.routers.test_posts.`**`test_create_comment`**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*, *confirmed_user_fixture: dict*, *logged_in_token_fixture: str*)

Test creating a comment on a post.

Verifies that a comment can be created and the response contains expected fields.

`async storeapi.tests.routers.test_posts.`**`test_create_post`**(*async_client_fixture: AsyncClient*, *confirmed_user_fixture: dict*, *logged_in_token_fixture: str*)

Test creating a new post.

Verifies that a post can be created and the response contains expected fields.

`async storeapi.tests.routers.test_posts.`**`test_create_post_expired_token`**(*async_client_fixture: AsyncClient*, *confirmed_user_fixture: dict*, *mocker*)

Test creating a post with an expired token.

Verifies that the request is rejected with a 401 Unauthorized response.

`async storeapi.tests.routers.test_posts.`**`test_create_post_missing_data`**(*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*)

Test creating a post with missing required data.

Verifies that the request is rejected with a 422 Unprocessable Entity response.

async storeapi.tests.routers.test_posts.**test_create_post_with_prompt**(*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*, *mocker*)

Test creating a post with an image generation prompt.

Verifies that the image generation task is called with the correct parameters.

async storeapi.tests.routers.test_posts.**test_get_all_posts**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*)

Test retrieving all posts.

Verifies that all posts can be retrieved and the response includes like counts.

async storeapi.tests.routers.test_posts.**test_get_all_posts_sort_likes**(*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*)

Test sorting posts by like count.

Verifies that posts are sorted by most likes when the 'most_likes' sorting is specified.

async storeapi.tests.routers.test_posts.**test_get_all_posts_sorting**(*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*, *sorting: str*, *expected_order: list[int]*)

Test sorting posts by creation time.

Parameters

- **sorting** – Sort direction ('new' or 'old')
- **expected_order** – Expected order of post IDs after sorting

Verifies that posts are sorted as expected based on the sorting parameter.

async storeapi.tests.routers.test_posts.**test_get_all_posts_wrong_sorting**(*async_client_fixture: AsyncClient*)

Test using an invalid sorting parameter.

Verifies that the request is rejected with a 422 Unprocessable Entity response.

async storeapi.tests.routers.test_posts.**test_get_comments_on_post**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*, *created_comment_fixture: dict*)

Test retrieving comments for a specific post.

Verifies that all comments for a post can be retrieved.

async storeapi.tests.routers.test_posts.**test_get_comments_on_post_empty**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*)

Test retrieving comments for a post that has no comments.

Verifies that an empty list is returned when a post has no comments.

async storeapi.tests.routers.test_posts.**test_get_missing_post_with_comments**(*async_client_fixture: AsyncClient*)

Test retrieving a non-existent post.

Verifies that a 404 Not Found response is returned when a post doesn't exist.

async storeapi.tests.routers.test_posts.**test_get_post_with_comments**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*, *created_comment_fixture: dict*)

Test retrieving a post with its comments.

Verifies that a post can be retrieved with its comments and like count.

async storeapi.tests.routers.test_posts.**test_like_post**(*async_client_fixture: AsyncClient*, *created_post_fixture: dict*, *logged_in_token_fixture: str*)

Test liking a post.

Verifies that a like can be created and the response contains expected fields.

Upload router test module. Tests file upload functionality including image uploads, temporary file handling, and integration with external storage services.

storeapi.tests.routers.test_upload.**aiofiles_mock_open**(*mocker*, *fs*)

Mock aiofiles.open to use the pyfakefs filesystem.

Parameters

- **mocker** – pytest-mock fixture
- **fs** – PyFakeFS fixture for file system operations

Returns

The mocked open function

Return type

MagicMock

async storeapi.tests.routers.test_upload.**call_upload_endpoint**(*async_client_fixture: AsyncClient*, *token: str*, *sample_image: Path*)

Helper function to call the upload endpoint.

Parameters

- **async_client_fixture** – HTTP client for making requests
- **token** – Authentication token
- **sample_image** – Path to the image file to upload

Returns

The HTTP response from the upload endpoint

Return type

Response

`storeapi.tests.routers.test_upload.`**`mock_b2_upload_file`**`(`*mocker*`)`

Mock the B2 file upload functionality.

Parameters
`mocker` – pytest-mock fixture

Returns
The mocked upload function

Return type
MagicMock

`storeapi.tests.routers.test_upload.`**`sample_image`**`(`*fs*`)` → Path

Create a sample image file for testing.

Parameters
`fs` – PyFakeFS fixture for file system operations

Returns
Path to the created sample image

Return type
pathlib.Path

`async` `storeapi.tests.routers.test_upload.`**`test_temp_file_remove_after_upload`**`(`*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*, *sample_image: Path*, *mocker*`)`

Test that temporary files are removed after upload.

Verifies that the temporary file created during upload is properly cleaned up.

`async` `storeapi.tests.routers.test_upload.`**`test_upload_image`**`(`*async_client_fixture: AsyncClient*, *logged_in_token_fixture: str*, *sample_image: Path*`)`

Test uploading an image file.

Verifies that the image is uploaded successfully and the correct response is returned.

User router test module. Tests user registration, confirmation, and authentication functionality for the Store API application.

`async` `storeapi.tests.routers.test_user.`**`register_user`**`(`*async_client_fixture: AsyncClient*, *email: str*, *password: str*`)` → Response

Helper function to register a user.

Parameters

- `async_client_fixture` – HTTP client for making requests
- `email` – User's email address
- `password` – User's password

Returns
HTTP response from the registration endpoint

Return type
Response

async storeapi.tests.routers.test_user.**test_confirm_user**(*async_client_fixture: AsyncClient*, *mocker*)

Test confirming a user's registration.

Verifies that a user can confirm their registration by following the confirmation URL sent via email.

async storeapi.tests.routers.test_user.**test_confirm_user_expired_token**(*async_client_fixture: AsyncClient*, *mocker*)

Test confirming registration with an expired token.

Verifies that an expired confirmation token is rejected with the appropriate error message.

async storeapi.tests.routers.test_user.**test_confirm_user_invalid_token**(*async_client_fixture: AsyncClient*)

Test confirming registration with an invalid token.

Verifies that an invalid confirmation token is rejected with the appropriate error message.

async storeapi.tests.routers.test_user.**test_login_user**(*async_client_fixture: AsyncClient*, *confirmed_user_fixture: dict*) → None

Test successful user login.

Verifies that a confirmed user can log in successfully and receive an access token.

async storeapi.tests.routers.test_user.**test_login_user_not_confirmed**(*async_client_fixture: AsyncClient*, *registered_user_fixture: dict*) → None

Test logging in with an unconfirmed user.

Verifies that attempting to login with an unconfirmed account returns an unauthorized response with the appropriate error message.

async storeapi.tests.routers.test_user.**test_login_user_not_exists**(*async_client_fixture: AsyncClient*) → None

Test logging in with a non-existent user.

Verifies that attempting to login with a non-existent email returns an unauthorized response.

async storeapi.tests.routers.test_user.**test_register_user**(*async_client_fixture: AsyncClient*) → None

Test registering a new user.

Verifies that a user can be registered successfully and the response contains the expected message.

async storeapi.tests.routers.test_user.**test_register_user_already_exists**(*async_client_fixture: AsyncClient*, *registered_user_fixture: dict*) → None

Test registering a user with an email that already exists.

Verifies that attempting to register with an existing email returns a conflict response with the appropriate error message.

PYTHON MODULE INDEX

INDEX