### All methods and fields in the system have public access by default.

### 1. ****JwtModule****

* **Attributes**:
* providers: JwtService — service for working with JWT.
* exports: JwtService — exports JwtService.
* **Methods**:
* register(): void — registers the module.

### 2. ****JwtService****

* **Methods**:
* signAsync(payload: JwtPayload, time: int): Tokens — asynchronous method to create a JWT token.

### 3. ****AuthModule****

* **Attributes**:
* imports: UsersModule, JwtModule — modules that are imported.
* providers: AuthService — service for authentication.
* controllers: AuthController — controller for handling authentication requests.

### 4. ****AuthController****

* **Methods**:
* register(userData: UserDTO): void — registers a new user.
* login(credentials: Credentials): Tokens — authenticates a user.
* logout(userId: ObjectId): void — logs the user out.
* refreshToken(rt: String): Tokens — refreshes the token.
* sso(method: SSOMethods): Tokens — Single Sign-On (SSO) authentication (e.g., Facebook, Google).

### 5. ****AuthService****

* **Methods**:
* register(data: UserDTO): User — registers a user.
* login(credentials: Credentials): Tokens — user login.
* logout(userId: ObjectId): void — logs the user out.
* refreshToken(rt: String): Tokens — refreshes the token.
* sso(method: SSOMethods): Tokens — authenticates via SSO.
* hashToken(token: String): String — hashes the token.
* compareTokens(token: String, hash: String): bool — compares the token with its hash.
* generateTokens(data: JwtPayload): Tokens — generates tokens.

### 6. ****SSOMethods (enum)****

* **Elements**:
* FACEBOOK
* GOOGLE
* APPLE

### 7. ****UserDTO****

* **Attributes**:
* name: String — user's name.
* email: String — user's email.
* password: String — user's password.

### 8. ****Credentials****

* **Attributes**:
* email: String — email address.
* password: String — password.

### 9. ****Tokens****

* **Attributes**:
* accessToken: String — access token.
* refreshToken: String — refresh token.

### 10. ****Role (enum)****

* **Elements**:
* OWNER
* ADMIN
* EDITOR
* USER
* GUEST

### 11. ****JwtPayload****

* **Attributes**:
* id: String — user's ID.
* email: String — user's email.
* role: Role — user's role.

### 12. ****ErrorCodes (enum)****

* **Elements**:
* USER\_NOT\_FOUND, INVALID\_CREDENTIALS, BOOK\_NOT\_FOUND, PAYMENT\_FAILED, and other error codes related to the system.

### 13. ****BooksModule****

* **Attributes**:
* imports: BooksSchema, UsersModule — modules being imported.
* providers: BooksService — service for handling book operations.
* controllers: BooksController — controller for book-related requests.
* exports: BooksService — exports the book service.

### 14. ****BooksController****

* **Methods**:
* getAllBooks(authorId: ObjectId): List<BookSchema> — retrieves all books by a specific author.
* searchBooks(searchParam: String): List<BookSchema> — searches for books based on a parameter.
* getBook(bookId: ObjectId): BookSchema — retrieves a book by its ID.
* suggestBook(bookData: BookDTO): void — suggests a new book.
* previewBook(bookId: ObjectId): BookSchema — previews a book.
* publishBook(bookId: ObjectId): void — publishes a book.
* rejectBook(bookId: ObjectId): void — rejects a book.
* updateBook(bookId: ObjectId, bookData: BookDTO): void — updates book data.
* deleteBook(bookId: ObjectId): void — deletes a book.
* buyBook(bookId: ObjectId): BookSchema — purchases a book.
* getBoughtBooks(userId: ObjectId): List<BookSchema> — retrieves the books bought by the user.
* addAudioVersion(audio: mp3): BookSchema — adds an audio version of the book.
* listenAudioBook(bookId: ObjectId): mp3 — listens to the audiobook.
* downloadBook(type: FileType): File — downloads the book in a specific file format.
* addBookToList(type: ListType, bookId: ObjectId): void — adds a book to a list.
* deleteBookFromList(type: ListType, bookId: ObjectId): void — removes a book from a list.

### 15. ****BooksService****

* **Methods**:
* getAllBooks(authorId: ObjectId): List<BookSchema> — retrieves all books by an author.
* searchBooks(searchParam: String): List<BookSchema> — searches for books.
* getBook(bookId: ObjectId): BookSchema — retrieves a book by ID.
* suggestBook(bookData: BookDTO): void — suggests a new book.
* previewBook(bookId: ObjectId): BookSchema — previews a book.
* publishBook(bookId: ObjectId): void — publishes a book.
* rejectBook(bookId: ObjectId): void — rejects a book.
* updateBook(bookId: ObjectId, data: BookDTO): void — updates a book's details.
* deleteBook(bookId: ObjectId): void — deletes a book.
* buyBook(bookId: ObjectId): BookSchema — purchases a book.
* getBoughtBooks(userId: ObjectId): List<BookSchema> — retrieves the books bought by the user.
* addAudioVersion(audio: mp3): BookSchema — adds an audio version of the book.
* listenAudioBook(bookId: ObjectId): mp3 — plays the audiobook.
* generateBookFile(type: FileType): File — generates a book file in the specified format.
* addBookToList(type: ListType, bookId: ObjectId): void — adds a book to a list.
* deleteBookFromList(type: ListType, bookId: ObjectId): void — removes a book from a list.

### 16. ****BookSchema****

* **Attributes**:
* id: ObjectId — unique identifier for the book.
* title: String — book title.
* author: String — book author.
* genre: String — book genre.
* cover: String — book cover.
* audioVersion: mp3 — audiobook version.
* isPublished: bool — whether the book is published.
* text: String — book text.
* bookmarks: List<ObjectId> — list of bookmarks associated with the book.
* isPaid: bool — whether the book is paid.
* price: float — book price.

### 17. ****BookDTO****

* **Attributes**:
* title: String — title of the book.
* author: String — author of the book.
* genre: String — genre of the book.
* cover: String — cover image of the book.
* audioVersion: mp3 — audio version of the book.
* text: String — text of the book.
* isPaid: bool — whether the book is paid.
* price: float — price of the book.

18. **FileType (enum)**

* **Elements**:
* EPUB — eBook format.
* FB2 — FictionBook format.
* TXT — plain text format.
* PDF — Portable Document Format.

### 19. ****BookmarksModule****

* **Attributes**:
* imports: BookmarksSchema, UsersModule, BooksModule — modules being imported.
* providers: BookmarksService — service for handling bookmark-related operations.
* controllers: BookmarksController — controller for bookmark-related requests.

### 20. ****BookmarksController****

* **Methods**:
* createBookmark(params: BookmarkDTO, bookId: ObjectId): BookmarksSchema — creates a bookmark.
* editBookmark(params: BookmarkDTO, bookmarkId: ObjectId): BookmarksSchema — edits an existing bookmark.
* deleteBookmark(bookmarkId: ObjectId): void — deletes a bookmark.
* getBookmarkById(bookmarkId: ObjectId): BookmarksSchema — retrieves a bookmark by its ID.
* getBookmarksByUserId(userId: ObjectId): List<BookmarksSchema> — retrieves bookmarks for a user.
* getBookmarksByBookId(bookId: ObjectId): List<BookmarksSchema> — retrieves bookmarks for a specific book.

### 21. ****BookmarksService****

* **Methods**:
* createBookmark(params: BookmarkDTO, bookId: ObjectId): BookmarksSchema — creates a bookmark.
* editBookmark(params: BookmarkDTO, bookmarkId: ObjectId): BookmarksSchema — edits a bookmark.
* deleteBookmark(bookmarkId: ObjectId): void — deletes a bookmark.
* getBookmarkById(bookmarkId: ObjectId): BookmarksSchema — retrieves a bookmark by ID.
* getBookmarksByUserId(userId: ObjectId): List<BookmarksSchema> — retrieves all bookmarks for a user.
* getBookmarksByBookId(bookId: ObjectId): List<BookmarksSchema> — retrieves all bookmarks for a specific book.

### 22. ****BookmarksSchema****

* **Attributes**:
* id: ObjectId — unique identifier for the bookmark.
* bookId: ObjectId — ID of the book the bookmark belongs to.
* userId: ObjectId — ID of the user who created the bookmark.
* indexPage: int — page number of the bookmark.
* color: String — color of the bookmark (used for categorization or organization).
* text: String — text associated with the bookmark.

### 23. ****BookmarkDTO****

* **Attributes**:
* color: String — color of the bookmark.
* text: String — text associated with the bookmark.

### 24. ****ReviewsModule****

* **Attributes**:
* imports: ReviewsSchema, BooksModule — modules being imported.
* providers: ReviewsService — service for handling reviews.
* controllers: ReviewsController — controller for review-related requests.

### 25. ****ReviewsController****

* **Methods**:
* getReviews(bookId: ObjectId): List<ReviewsSchema> — retrieves all reviews for a specific book.
* addReview(bookId: ObjectId, reviewData: ReviewDTO): void — adds a new review for a book.
* deleteReview(reviewId: ObjectId): void — deletes a review.
* editReview(reviewId: ObjectId, reviewData: ReviewDTO): void — edits an existing review.

### 26. ****ReviewsService****

* **Methods**:
* fetchReviews(bookId: ObjectId): List<ReviewsSchema> — fetches all reviews for a given book.
* createReview(bookId: ObjectId, reviewData: ReviewDTO): void — creates a review for a book.
* removeReview(reviewId: ObjectId): void — removes a review.
* editReview(reviewId: ObjectId, reviewData: ReviewDTO): void — edits an existing review.

### 27. ****ReviewsSchema****

* **Attributes**:
* id: ObjectId — unique identifier for the review.
* bookId: ObjectId — ID of the book being reviewed.
* userId: ObjectId — ID of the user who posted the review.
* rating: int — rating given to the book (e.g., 1–5).
* text: String — textual content of the review.

### 28. ****ReviewDTO****

* **Attributes**:
* rating: int — rating for the book.
* text: String — review text.

### 29. ****PaymentModule****

* **Attributes**:
* imports: PaymentSchema, UsersModule — modules being imported.
* providers: PaymentService — service for handling payments.
* controllers: PaymentController — controller for payment-related requests.

### 30. ****PaymentController****

* **Methods**:
* processPayment(paymentData: PaymentDTO): void — processes a payment.
* getPaymentDetails(paymentId: ObjectId): PaymentSchema — retrieves payment details by ID.

### 31. ****PaymentService****

* **Methods**:
* processPayment(paymentData: PaymentDTO): void — processes the payment transaction.
* fetchPaymentDetails(paymentId: ObjectId): PaymentSchema — retrieves payment details by ID.

### 32. ****PaymentSchema****

* **Attributes**:
* id: ObjectId — unique identifier for the payment.
* userId: ObjectId — ID of the user making the payment.
* amount: float — payment amount.
* cardNumber: String — card number used for the payment.
* date: Date — payment date.

### 33. ****PaymentDTO****

* **Attributes**:
* userId: ObjectId — user ID for the payment.
* amount: float — amount for the payment.
* cardNumber: String — card number used.

### 34. ****UsersModule****

* **Attributes**:
* imports: UsersSchema, BooksModule — modules being imported.
* providers: UsersService — service for handling user operations.
* controllers: UsersController — controller for user-related requests.
* exports: UsersService — exports the user service.

### 35. ****UsersController****

* **Methods**:
* getUser(id: ObjectId): UsersSchema — retrieves a user by ID.
* getUsers(ids: List<ObjectId>): List<UsersSchema> — retrieves multiple users by their IDs.
* changeInfo(userData: UserDTO): UserSchema — updates user information.
* changeRole(id: ObjectId, role: Role): UsersSchema — changes a user's role.
* deleteUser(id: ObjectId): void — deletes a user.
* blockUser(id: ObjectId): void — blocks a user.
* startSubscription(cardNumber: String): void — starts a user subscription.
* stopSubscription(): void — stops the user subscription.
* becomeEditor(data: EditorForm): void — applies for the role of editor.
* allowEditor(userId: ObjectId): void — grants editor rights to a user.
* rejectEditor(userId: ObjectId): void — rejects editor rights for a user.
* getList(type: ListType): List<ObjectId> — retrieves a list of books based on the user's list type (e.g., liked, want, etc.).

### 36. ****UsersService****

* **Methods**:
* createUser(userData: UserDTO): UsersSchema — creates a new user.
* getUserById(id: ObjectId): UsersSchema — retrieves a user by ID.
* getUserByEmail(email: String): UsersSchema — retrieves a user by email.
* getUsers(ids: List<ObjectId>): List<UsersSchema> — retrieves multiple users by their IDs.
* deleteUser(id: ObjectId): void — deletes a user.
* blockUser(id: ObjectId): void — blocks a user.
* changeInfo(userData: UserDTO): UserSchema — updates user information.
* changeRole(id: ObjectId, role: Role): UsersSchema — changes a user's role.
* addBoughtBook(bookId: ObjectId): void — adds a book to the list of bought books.
* getBoughtBooks(userId: ObjectId): List<ObjectId> — retrieves a list of books purchased by a user.
* startSubscription(cardNumber: String): void — starts a subscription for a user.
* stopSubscription(): void — stops a user's subscription.
* becomeEditor(data: EditorForm): void — applies for an editor position.
* allowEditor(userId: ObjectId): void — grants editor status to a user.
* rejectEditor(userId: ObjectId): void — denies editor status to a user.
* addBookToList(type: ListType, bookId: ObjectId): void — adds a book to the user's list.
* deleteBookFromList(type: ListType, bookId: ObjectId): void — removes a book from the user's list.
* getList(type: ListType): List<ObjectId> — retrieves a user's book list of a given type (e.g., liked, want, etc.).

### 37. ****UsersSchema****

* **Attributes**:
* id: ObjectId — unique identifier for the user.
* name: String — user's name.
* email: String — user's email.
* password: String — user's password.
* passwordHash: String — hashed version of the password.
* rtHash: String — hashed version of the refresh token.
* role: String — user role (e.g., OWNER, ADMIN, etc.).
* balance: float — user's balance (e.g., money or credits).
* subscription: bool — whether the user has a subscription.
* boughtBooks: List<ObjectId> — list of books the user has bought.
* liked: List<ObjectId> — list of books the user has liked.
* want: List<ObjectId> — list of books the user wants to read.
* reading: List<ObjectId> — list of books the user is currently reading.
* read: List<ObjectId> — list of books the user has finished reading.

### 38. ****EditorForm****

* **Attributes**:
* alias: String — The pseudonym or pen name that the user wants to use as an editor.
* mainGenre: String — The main literary genre the user specializes in as an editor (e.g., Fiction, Non-fiction, Science Fiction, etc.).
* whyBecomeEditor: String — The motivation or reasoning behind the user's desire to become an editor.
* experience: int — The number of years of experience the user has as an editor.

### ListType Enum****Attributes:****

* **LIKED**: Represents books that the user likes or favorites. Users can add books to this list to indicate their interest in them.
* **WANT**: Represents books that the user wants to read in the future. These books are often added to a "wishlist."
* **READ**: Represents books that the user has already read.
* **READING**: Represents books that the user is currently reading.

### EditorForm Entity

#### ****Attributes:****

* **alias** (String): The alias or nickname the user prefers to be known by as an editor.
* **mainGenre** (String): The primary genre or type of books the user specializes in or is most passionate about.
* **whyBecomeEditor** (String): A brief explanation from the user on why they want to become an editor.
* **experience** (int): The number of years or amount of experience the user has in the editorial field, or related activities.

### DatabaseModule Class****Attributes:****

* **uri** (String): The connection string or URI to the database. It contains the details needed to establish a connection (e.g., host, port, database name).

#### ****Methods:****

* **connect()** (void): Establishes a connection to the database using the provided uri.
* **close()** (void): Closes the connection to the database, ensuring that any active sessions or transactions are properly ended.

### MongoDB Class (within the Database package)

#### ****Attributes:****

* **uri** (String): The connection string or URI used to connect to the MongoDB instance.

#### ****Methods:****

* **connect()** (void): Connects to the MongoDB database using the uri. It opens a connection that allows the system to read and write data.
* **close()** (void): Closes the MongoDB connection, ensuring that no resources are left open after database operations are completed.