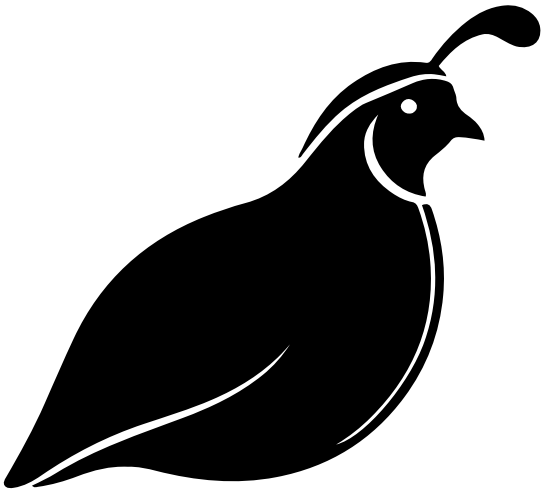
QUAiL

Developer

Guide



Version:

Beta v1.0.0

About This Guide

This document is the developer guide for the QUAiL web-application. It is intended for developers who will be maintaining and extending the project. The guide is organized into sections that correspond to the major components of the application's architecture: the Backend, the Frontend, and the Statistics Microservice. Every section includes a detailed overview of the directory structure, the roles and responsibilities of individual files, API route specifications, and relevant data schemas. The goal is to help developers understand the organization of the codebase, trace functionality efficiently, and ensure consistent development practices.

Note to Future Authors: To ensure the accuracy and relevance of this documentation, it is crucial that any significant changes to the codebase are reflected in this guide. Please update the version number and date on the cover page and in the headers to correspond with new software releases.

Version	Author	Date
QUAiL Developer Guide v1.0.0	Saket Sontakke	18-Sep-25

Table of Contents

1. Backend

The backend is a robust Node.js application built with the Express.js framework, responsible for handling business logic, user authentication, data persistence with MongoDB, and acting as an API gateway to the Python microservice for statistical analysis.

1.1. backend Directory

The project follows a standard Model-View-Controller (MVC) architectural pattern, which is reflected in its directory structure. This organization promotes a clear separation of concerns, making the codebase easier to navigate and maintain.

```
backend/
│   └── server.js
├── src/
│   ├── models/
│   │   ├── Project.js
│   │   └── Users.js
│   ├── utils/
│   │   └── sendEmail.js
│   ├── controllers/
│   │   ├── authController.js
│   │   ├── projectController.js
│   │   └── statsController.js
│   ├── routes/
│   │   ├── authRoutes.js
│   │   ├── projectManagementRoutes.js
│   │   ├── projectFileManagementRoutes.js
│   │   ├── projectAnnotationRoutes.js
│   │   ├── projectExportRoutes.js
│   │   ├── projectRoutes.js
│   │   └── statsRoutes.js
├── uploads/
│   ├── audio/
│   └── text/
├── __tests__/
├── node_modules/
├── .Dockerignore
├── .env
├── .env.test
├── .gitignore
├── Dockerfile
├── jest.config.js
├── package-lock.json
└── package.json
```

1.2. Commands

This section details the primary commands used to run, develop, and test the backend application. These commands are defined in the scripts section of the package.json file.

1.2.1. npm start

- Purpose: To start the application for a production environment.
- Description: This command runs the main application file (src/server.js) using the standard Node.js runtime. It's the essential command for deploying and running the server in a live setting. It does not automatically restart on file changes, making it stable and efficient for production use.
- Script: "start": "node src/server.js"

1.2.2. npm run dev

- Purpose: To start the application in development mode with hot-reloading.
- Description: This command uses nodemon to execute the application. Nodemon is a tool that automatically monitors for any file changes in the source directory and restarts the server upon detection. This is extremely useful during development as it removes the need to manually stop and restart the server after making code changes.
- Script: "dev": "nodemon src/server.js"

1.2.3. npm test

- Purpose: To execute the project's automated test suite.
- Description: This command runs all tests using the Jest testing framework. The script is configured to perform several key actions:
    - cross-env NODE_ENV=test: It sets the environment variable NODE_ENV to 'test'. As described in the server.js documentation, this special mode prevents the application from connecting to the live database or starting the web server listener, creating an isolated environment for testing.
    - NODE_NO_WARNINGS=1: This suppresses certain experimental feature warnings from Node.js.
    - node --experimental-vm-modules: This flag enables ES module support within Jest's test environment.
    - jest.js --runInBand: This executes the Jest test runner. The --runInBand flag ensures that tests run serially in the same process, which can prevent issues with tests that access a shared resource, like an in-memory database.
- Script: "test": "cross-env NODE_ENV=test NODE_NO_WARNINGS=1 node --experimental-vm-modules node_modules/jest/bin/jest.js --runInBand"

1.3. Server Setup

1.3.1. server.js

Purpose	Initializes the Express application, configures middleware, mounts API routes, connects to the database, and starts the server.	
Dependencies	**Internal**	./routes/authRoutes.js ./routes/projectRoutes.js ./routes/statsRoutes.js
	External	express mongoose cors dotenv
Key Components	Middleware Setup: Configures the application to use cors for cross-origin requests and express.json() for parsing JSON request bodies. Static File Serving: Serves static assets from the ../uploads directory via the /uploads route. Route Mounting: Mounts separate router files (authRoutes, projectRoutes, statsRoutes) to their respective base API paths. Database Connection & Server Start: Conditionally connects to MongoDB using Mongoose and starts the Express server, but skips this process if NODE_ENV is set to 'test'.	
Usage	This file acts as the main router, delegating endpoint handling to other files based on the URL prefix. POST /api/auth/...: All authentication-related requests are handled by authRoutes.js. GET /api/projects/...: All project-related requests are handled by projectRoutes.js. GET /api/stats/...: All statistics-related requests are handled by statsRoutes.js. GET /uploads/...: Serves static files from the server's uploads directory.	
Data Schema	N/A	

.env Configuration	PORT: The port number on which the server will listen (defaults to 5000). MONGO_URI: The connection string for the MongoDB database. NODE_ENV: The runtime environment (e.g., 'development', 'production', 'test'). If set to 'test', the database connection and server listener are disabled.
Error Handling	▪ Database Connection Failure: Catches and logs errors that occur during the initial connection attempt to the MongoDB database, preventing the server from starting.

1.4. Database Models

1.4.1. Users.js

Purpose	Defines the Mongoose schema for a User, storing user credentials and information required for authentication and password reset functionality.	
Dependencies	**Internal**	N/A
	External	mongoose
Key Components	userSchema: The Mongoose schema that outlines the structure of a user document in the database. User Model: The compiled Mongoose model, exported to allow other parts of the application to interact with the users collection.	
Usage	This is a model file and does not define any API endpoints. It is imported and used by controller and service files to perform CRUD (Create, Read, Update, Delete) operations on the MongoDB database.	
Data Schema	This file defines the data structure for a document in the users collection. ▪ name (String): The full name of the user. ▪ email (String, unique): The user's unique email address, used for login. ▪ password (String): The user's hashed password. ▪ resetToken (String): A temporary token for the password reset process. ▪ resetTokenExpiry (Date): The expiration date for the resetToken. ▪ createdAt (Date): Automatically added by Mongoose via timestamps, records when the user was created. ▪ updatedAt (Date): Automatically added by Mongoose via timestamps, records the last update time.	

.env Configuration	N/A
Error Handling	▪ Schema Validation: Mongoose automatically validates data against this schema upon save. For example, it enforces the unique constraint on the email field. ▪ Duplicate Key Error: If an attempt is made to create a user with an email that already exists, MongoDB will return a duplicate key error, which must be caught and handled by the controller logic that uses this model.

1.4.2. Project.js

Purpose	Defines the Mongoose database schema for the Project collection, which serves as the central container for all data related to a single qualitative analysis project.	
Dependencies	**Internal**	N/A
	External	mongoose
Key Components	projectSchema: The main schema that aggregates all project-related data, including files, codes, memos, and highlights as embedded sub-documents. Sub-document Schemas: ▪ codeDefinitionSchema: Defines the structure for a user-created code/tag. ▪ importedFileSchema: Defines the structure for an uploaded file's content and metadata. ▪ codedSegmentSchema: Defines a segment of text that has been linked to a code. ▪ inlineHighlightSchema: Defines a simple, colored highlight on a text segment. ▪ memoSchema: Defines a user-written note attached to a text segment. Project Model: The compiled Mongoose model exported for use in other parts of the application to interact with the projects collection in MongoDB.	
Usage	This is a model file and does not define any API endpoints. It is imported and used by controller and service files to perform CRUD (Create, Read, Update, Delete) operations on the MongoDB database.	
Data Schema	This file defines the core data structure for a Project and its nested components.	

	Project Collection Schema: ▪ name (String, required): The title of the project. ▪ owner (ObjectId, ref: 'User', required): The user who created the project. ▪ importedFiles (Array of importedFileSchema): A list of all documents within the project. ▪ codeDefinitions (Array of codeDefinitionSchema): A list of all codes created for the project. ▪ codedSegments (Array of codedSegmentSchema): A list of all coded text snippets. ▪ inlineHighlights (Array of inlineHighlightSchema): A list of all highlighted text snippets. ▪ memos (Array of memoSchema): A list of all memos. importedFileSchema Sub-document: ▪ name (String, required): The original filename. ▪ content (String, required): The text content of the file. ▪ sourceType (String, enum: ['text', 'audio']): The type of the source file. ▪ properties (Map of String): Key-value metadata for the file. codeDefinitionSchema Sub-document: ▪ name (String, required): The name of the code. ▪ description (String): An optional explanation of the code. ▪ color (String): A hex color code for the UI.
.env Configuration	N/A
Error Handling	▪ Schema validation is handled automatically by Mongoose based on the definitions provided (e.g., required: true, enum constraints). If an operation attempts to save a Project document that violates the schema, Mongoose will throw a ValidationError. This error is expected to be caught and handled by the controller or service logic that uses this model.

1.5. Utilities

1.5.1. sendEmail.js

Purpose	Provides a reusable utility function for sending emails via a pre-configured Gmail SMTP transporter.	
Dependencies	**Internal**	N/A
	External	nodemailer
Key Components	sendEmail(to, subject, text): The sole exported asynchronous function that configures and sends an email using credentials from environment variables.	
Usage	This is a utility file and does not define any API endpoints. It is designed to be imported and used by other services or controllers that require email-sending functionality.	
Data Schema	This function does not interact with request bodies. Its data interface is defined by its function parameters: ▪ to (String, required): The email address of the recipient. ▪ subject (String, required): The subject line of the email. ▪ text (String, required): The plain text content of the email body.	
.env Configuration	EMAIL_USER: The username (email address) for the Gmail account used to send emails. EMAIL_PASS: The password or an app-specific password for the EMAIL_USER account.	
Error Handling	The sendEmail function is asynchronous and does not contain an internal try...catch block. If nodemailer fails to send the email (e.g., due to invalid credentials, network issues, or service unavailability), it will throw an error. The calling function is responsible for catching and handling this error.	

1.6. Controllers

1.6.1. authController.js

Purpose	Handles user registration, login, password management (forgot/reset), and provides middleware for protecting routes with JSON Web Tokens (JWT).	
Dependencies	**Internal**	../models/Users.js ../utils/sendEmail.js
	External	bcryptjs jsonwebtoken
Key Components	protect(req, res, next): Middleware that verifies a JWT from the request's Authorization header, attaching the authenticated user to req.user if valid. registerUser(req, res): Creates a new user, hashes their password, and returns a JWT for immediate login. loginUser(req, res): Authenticates a user's credentials and issues a JWT upon successful validation. forgotPassword(req, res): Generates a password reset token and sends a reset link to the user's email address. resetPassword(req, res): Verifies a reset token and updates the user's password with a new one.	
Usage	POST /api/auth/register ▪ Description: Registers a new user account. ▪ Body: { "name": "Test User", "email": "test@example.com", "password": "password123" } ▪ Success Response: 201 Created with { "token": "JWT_TOKEN", "message": "User created successfully" } ▪ Error Response: 400 Bad Request if the user already exists. POST /api/auth/login ▪ Description: Logs in an existing user. ▪ Body: { "email": "test@example.com", "password": "password123" } ▪ Success Response: 200 OK with { "token": "JWT_TOKEN", "user": { "name": "Test User", "email": "test@example.com" } } ▪ Error Response: 400 Bad Request for invalid credentials. POST /api/auth/forgot-password	

	▪ Description: Initiates the password reset process. ▪ Body: { "email": "test@example.com" } ▪ Success Response: 200 OK with a generic confirmation message. POST /api/auth/reset-password/:token ▪ Description: Sets a new password using a token from the reset link. ▪ URL Params: token (The JWT received in the reset email). ▪ Body: { "password": "newPassword456" } ▪ Success Response: 200 OK with { "message": "Password reset successful" } ▪ Error Response: 400 Bad Request if the token is invalid or expired. protect Middleware ▪ Description: To use this on a route, add it before your controller function. It expects an Authorization header. ▪ Header: Authorization: Bearer JWT_TOKEN
Data Schema	User Registration Body: ▪ name (String, required) ▪ email (String, required) ▪ password (String, required) User Login Body: ▪ email (String, required) ▪ password (String, required) ▪ Forgot Password Body: ▪ email (String, required) Reset Password Body: ▪ password (String, required)
.env Configuration	JWT_SECRET: The secret key used for signing and verifying JSON Web Tokens. CLIENT_URL: The base URL of the client-side application, used for constructing the password reset link sent via email. NODE_ENV: Determines if errors are logged to the console (e.g., in the resetPassword function).
Error Handling	▪ Returns a 401 Unauthorized status from the protect middleware if the token is missing, invalid, expired, or the associated user is not found. ▪ Returns a 400 Bad Request if registration fails due to a duplicate email.

	▪ Returns a 400 Bad Request on login if credentials are invalid. ▪ Returns a 400 Bad Request during password reset if the token is invalid/expired or the new password is the same as the old one. ▪ The forgotPassword endpoint always returns a 200 OK with a generic message to prevent attackers from discovering which emails are registered.

1.6.2. projectController.js

Purpose	Provides route protection middleware and controller logic for fetching user-owned project data.	
Dependencies	**Internal**	../models/Project.js
	External	jsonwebtoken
Key Components	requireAuth(req, res, next): An Express middleware that protects routes by validating a JSON Web Token from the Authorization header and attaching the user's ID to the request (req.userId). getProjectById(req, res): A controller that retrieves a single project from the database, ensuring the project exists and belongs to the authenticated user.	
Usage	GET /api/projects/:id ▪ Description: Fetches a single project by its unique ID. This route must be protected by the requireAuth middleware. ▪ Headers: Authorization: Bearer JWT_TOKEN ▪ URL Params: id (The MongoDB ObjectId of the project). ▪ Success Response: 200 OK with the full project JSON object. ▪ Error Response: 404 Not Found if the project does not exist or the user is not the owner. 500 Internal Server Error for database issues. requireAuth Middleware ▪ Description: This is not an endpoint but a middleware function to be used in a route definition before the final controller (e.g., router.get('/:id', requireAuth, getProjectById)). ▪ Error Response: 401 Unauthorized if the token is missing, malformed, or invalid.	
Data Schema	N/A	
.env Configuration	JWT_SECRET: The secret key required by the requireAuth middleware to verify the signature of the JSON Web Token.	

Error Handling	▪ Returns a 401 Unauthorized status if no token is provided or if the provided token is invalid. ▪ Returns a 404 Not Found status if a project with the given ID cannot be found for the authenticated user, preventing data leaks. ▪ Returns a 500 Internal Server Error and logs the error message if an unexpected issue occurs while querying the database.

1.6.3. statsController.js

Purpose	Handles requests for statistical analysis, specifically orchestrating Chi-Square tests by preparing data, validating assumptions, and calling an external Python microservice for computation.	
Dependencies	**Internal**	../models/Project.js
	External	axios
Key Components	runTest(req, res): The main exported controller that acts as a router. It validates the request, fetches project data, delegates to the appropriate data preparation function based on the test subtype, and either runs assumption checks or forwards the prepared data to the Python microservice. Data Preparation Helpers: A set of internal functions (prepareGofTest, prepareIndependenceTest, prepareHomogeneityTest) that transform raw project data into the contingency tables or frequency counts required for each specific Chi-Square test. applyCodeCombinations(...): A utility function that merges rows of a contingency table based on user-defined code groupings, allowing for more flexible analysis. Assumption Checking Helpers: A set of internal functions (performAssumptionChecks, checkGofAssumptions, checkContingencyAssumptions) that validate statistical assumptions, such as minimum expected cell frequencies, before running the full test.	
Usage	POST /api/stats/run-test ▪ Description: Executes a Chi-Square statistical test or validates its assumptions. The specific behavior and required data depend on the chiSquareSubtype. ▪ Body: (See Data Schema section for detailed structure) ▪ Success Response: 200 OK. ▪ If validateOnly: true, returns { "validationResults": { ... } }.	

	▪ Otherwise, returns a full statistical result object from the analysis service, e.g., { "statistic": 5.4, "pValue": 0.02, "df": 1, ... }. ▪ Error Response: 400 Bad Request for invalid input. 404 Not Found if the project ID is not found. 500 Internal Server Error for calculation failures or issues communicating with the analysis microservice.
Data Schema	The request body for the runTest endpoint has a common structure with subtype-specific fields. Common Fields: ▪ projectId (String, required): The ID of the project to analyze. ▪ testType (String, required): Must be 'chi-square'. ▪ chiSquareSubtype (String, required): One of 'goodness-of-fit', 'independence', 'homogeneity', or 'fishers-exact'. ▪ validateOnly (Boolean, optional): If true, the endpoint only performs and returns assumption checks. Subtype-Specific Fields: If chiSquareSubtype is 'goodness-of-fit': ▪ codes (Array of String, required): IDs of the codes to count. ▪ docList (Array of String, required): IDs of the documents to include. If chiSquareSubtype is 'independence': ▪ indepCodes (Array of String, required): Code IDs for the table rows. ▪ indepDocs (Array of String, required): Document IDs for the table columns. ▪ codeCombinations (Array of Object, optional): Groups of codes to merge. If chiSquareSubtype is 'homogeneity' or 'fishers-exact': ▪ homoCodes (Array of String, required): Code IDs for the table rows. ▪ homoDocGroups (Object, required): An object where keys are group names and values are arrays of document IDs. ▪ codeCombinations (Array of Object, optional): Groups of codes to merge.
.env Configuration	PYTHON_API_URL: The complete URL for the external Python microservice that performs the core statistical calculations.
Error Handling	▪ Validates testType and chiSquareSubtype at the beginning, returning a 400 Bad Request for invalid values. ▪ Returns a 404 Not Found if the provided projectId does not correspond to an existing project.

	▪ The main controller is wrapped in a try...catch block to handle unexpected errors gracefully. ▪ Catches errors from the axios request to the Python service and forwards the appropriate status code and message. ▪ Includes robust validation on the response from the Python service, checking for malformed JSON, invalid data structures, or null pValue to ensure data integrity.

1.7. Routes

1.7.1. authRoutes.js

Purpose	Defines and maps all API routes related to user authentication, delegating the request handling to the appropriate controller functions.	
Dependencies	**Internal**	../controllers/authController.js
	External	express
Key Components	express.Router(): The core component used to create a modular, mountable route handler for the authentication endpoints.	
Usage	This file defines the public-facing endpoints for the /api/auth path. POST /api/auth/register: Handles new user registration. POST /api/auth/login: Authenticates an existing user. POST /api/auth/forgot-password: Initiates the password reset process for a user. POST /api/auth/reset-password/:token: Completes the password reset process using a valid token.	
Data Schema	This router file defines the endpoints but does not directly process the request bodies. The linked controller functions in authController.js are responsible for handling the data schemas for each route.	
.env Configuration	N/A	
Error Handling	Error handling is not managed within this file. All request logic, including success and error responses, is delegated to and handled by the imported controller functions from authController.js.	

1.7.2. projectManagementRoutes.js

Purpose	Defines the core API routes for managing entire projects, including creating, reading, updating, deleting (CRUD), and copying projects for an authenticated user.	
Dependencies	**Internal**	../models/Project.js
	External	express mongoose
Key Components	This file implements controller logic directly within the route handlers: POST /create: Creates a new project after checking for existing projects with the same name. GET /my-projects: Fetches a list of all projects owned by the currently authenticated user. GET /:id: Fetches a single project by its ID, ensuring the user is the owner. PUT /:id: Updates a project's details, such as its name, after checking for name conflicts. DELETE /:id: Deletes a project and all its associated data. POST /:projectId/copy: Performs a deep copy of an existing project, creating a new, independent project with intelligently re-mapped internal ID references for all sub-documents.	
Usage	All endpoints are implicitly protected and require user authentication, as they rely on req.userId to scope database queries. POST /create ▪ Description: Creates a new project. ▪ Body: { "name": "New Project Name", "data": {} } ▪ Success Response: 201 Created with the new project object. ▪ Error Response: 409 Conflict if a project with that name already exists. GET /my-projects ▪ Description: Retrieves all projects for the logged-in user. ▪ Success Response: 200 OK with an array of project objects. GET /:id ▪ Description: Retrieves a single project by its ID. ▪ URL Params: id (The project's ID).	

	▪ Success Response: 200 OK with the project object. ▪ Error Response: 404 Not Found if the project doesn't exist or doesn't belong to the user. PUT /:id ▪ Description: Updates a project's name or data. ▪ URL Params: id (The project's ID). ▪ Body: { "name": "Updated Project Name" } ▪ Success Response: 200 OK with the updated project object. ▪ Error Response: 409 Conflict if the new name is already in use by another project. DELETE /:id ▪ Description: Deletes a project. ▪ URL Params: id (The project's ID). ▪ Success Response: 200 OK with { "message": "Deleted" }. POST /:projectId/copy ▪ Description: Creates a complete duplicate of an existing project. ▪ URL Params: projectId (The ID of the project to copy). ▪ Body: { "includeAnnotations": true } (Set to false to copy only the files without codes, memos, etc.). ▪ Success Response: 201 Created with the new (copied) project object.
Data Schema	Create Project Body: ▪ name (String, required) ▪ data (Object, optional) Update Project Body: ▪ name (String, optional) ▪ data (Object, optional) Copy Project Body: ▪ includeAnnotations (Boolean, required)
.env Configuration	N/A
Error Handling	▪ Authorization: All routes are scoped to the authenticated user via req.userId. Attempts to access or modify another user's project will result in a 404 Not Found error.

	▪ Name Conflicts: The create and update routes perform a case-insensitive check for duplicate project names and return a 409 Conflict to prevent duplicates. ▪ Resource Not Found: All routes that operate on a specific project ID will return a 404 Not Found if the project does not exist. ▪ Server Errors: All database operations are wrapped in try...catch blocks, and any unexpected failure will result in a 500 Internal Server Error.

1.7.3. projectFileManagementRoutes.js

Purpose	Defines all API routes to manage files within a project, including uploading and processing text, transcribing audio via an external service, and performing CRUD operations on imported files.	
Dependencies	**Internal**	../models/Project.js
	External	express multer mammoth axios dotenv
Key Components	Multer Configuration: ▪ fileStorage: A multer.diskStorage engine that saves uploaded files to disk, separating them by type (audio/text) and adding a timestamp to prevent name collisions. ▪ textUpload / audioUpload: Configured multer middleware for handling text and audio uploads with specific file type and size limit enforcement. ▪ handleMulterError: Middleware to gracefully handle errors from Multer, such as oversized files. Route Handlers: ▪ /files/stage: Processes an uploaded text file (.txt, .docx, .rtf), extracts its content, checks for name conflicts, and returns the processed text to the client for review without saving it to the database. ▪ /files/commit: Saves a prepared file (with its name and content) to the project's database.	

	▪ /import-audio: Manages the entire audio transcription pipeline: uploads a file, sends it to the AssemblyAI API, polls for the completed transcript, formats the result, and saves it to the project. ▪ /files/:fileId: A set of handlers for updating a file's content (PUT), renaming it (PUT /rename), and deleting it (DELETE), which also cleans up all associated data like codes and memos.
Usage	All endpoints are implicitly protected and require user authentication. POST /:projectId/files/stage ▪ Description: Uploads and processes a text file for review before committing. ▪ Request: multipart/form-data with a file field and optional splittingOption and overrideName fields. ▪ Success Response: 200 OK with { stagedFile: { name, content, sourceType } }. ▪ Error Response: 409 Conflict if filename exists; 413 Payload Too Large if file exceeds size limits. POST /:projectId/files/commit ▪ Description: Saves a new file's content to the project database. ▪ Body: { "name": "...", "content": "...", "sourceType": "..." } ▪ Success Response: 200 OK with the updated project object. POST /import-audio/:id ▪ Description: Uploads an audio file and replaces it with a generated transcript. ▪ Request: multipart/form-data with an audio field. ▪ Success Response: 200 OK with the updated project object containing the new transcript file. PUT /:projectId/files/:fileId/rename ▪ Description: Renames an existing file in the project. ▪ Body: { "name": "new-file-name.txt" } ▪ Success Response: 200 OK with the updated project object. PUT /:projectId/files/:fileId ▪ Description: Updates the text content of an existing file. ▪ Body: { "content": "Updated file content..." } ▪ Success Response: 200 OK with the updated project object. DELETE /:projectId/files/:fileId

	▪ Description: Deletes a file and all its associated data (codes, highlights, memos). ▪ Success Response: 200 OK with { message: "...", project: ... }.
Data Schema	Commit File Body: ▪ name (String, required) ▪ content (String, required) ▪ sourceType (String, optional): e.g., 'text', 'audio'. ▪ audioUrl (String, optional): A path to the associated audio file on the server. ▪ words (Array of Object, optional): Word-level timing and speaker data for transcripts. Rename File Body: ▪ name (String, required) Update File Content Body: ▪ content (String, required)
.env Configuration	NODE_ENV: Determines the base directory for file uploads (uploads for production/development, test_uploads for testing). ASSEMBLYAI_API_KEY: The API key for the AssemblyAI transcription service. The audio import feature will fail if this is not set.
Error Handling	▪ File Size Limits: Returns 413 Payload Too Large if an uploaded file exceeds the configured size limits (At the time of documentation-25MB for text/audio). ▪ File Type Validation: Multer's fileFilter rejects unsupported file types before they are processed. ▪ Name Conflicts: Returns 409 Conflict with a suggested new name if a user tries to upload or rename a file to a name that already exists in the project. ▪ External API Failures: The audio import route catches errors from the AssemblyAI API, logs detailed information, and returns a 500 Internal Server Error with relevant details. It also handles transcription timeouts. ▪ File System Cleanup: Ensures temporary files uploaded to the server are deleted after processing or in the event of an error. The delete route also removes stored audio files from the disk.

1.7.4. projectAnnotationRoutes.js

Purpose	Defines all API routes for performing CRUD (Create, Read, Update, Delete) and other complex operations on a project's sub-documents, including code definitions, coded segments, highlights, and memos.	
Dependencies	**Internal**	../models/Project.js
	External	express mongoose
Key Components	This file implements all controller logic directly within the route handlers, grouped by the type of data they manage: Code Definition Routes: Handlers for adding, updating, deleting, merging, and splitting the codes used for analysis. Coded Segment Routes: Handlers for creating, deleting, and re-assigning text segments that have been coded. Highlight Routes: Handlers for creating and deleting simple inline text highlights. Memo Routes: Handlers for creating, updating, and deleting user-written memos attached to text.	
Usage	All endpoints are nested under a project ID (e.g., /api/projects/:projectId/...) and require an authentication token, as they rely on req.userId to verify project ownership. Code Definitions ▪ POST /:projectId/code-definitions: Adds a new code. ▪ PUT /:projectId/code-definitions/:codeDefId: Updates an existing code. ▪ DELETE /:projectId/code-definitions/:codeDefId: Deletes a code and its associated segments. ▪ POST /:projectId/codes/merge: Merges multiple codes into one new code. ▪ POST /:projectId/codes/split: Splits one code into multiple new codes and reassigns segments. Coded Segments ▪ POST /:projectId/code: Creates a new coded segment. ▪ PUT /:projectId/code/:segmentId: Updates a segment (e.g., changes its code).	

	▪ DELETE /:projectId/code/:codeId: Deletes a single coded segment. Highlights ▪ POST /:projectId/highlight: Adds a new highlight. ▪ DELETE /:projectId/highlight/:highlightId: Deletes a single highlight. ▪ POST /:projectId/highlight/delete-bulk: Deletes multiple highlights at once. Memos ▪ POST /:projectId/memos: Adds a new memo. ▪ PUT /:projectId/memos/:memoId: Updates an existing memo. ▪ DELETE /:projectId/memos/:memoId: Deletes a memo.
Data Schema	Create Code Definition Body: ▪ name (String, required) ▪ description (String, optional) ▪ color (String, optional) Merge Codes Body: ▪ sourceCodeIds (Array of String, required) ▪ newCodeName (String, required) ▪ newCodeColor (String, required) Create Coded Segment Body: ▪ fileId, fileName, text (String, required) ▪ codeDefinitionId (String, required) ▪ startIndex, endIndex (Number, required) Bulk Delete Highlights Body: ▪ ids (Array of String, required) Create Memo Body: ▪ fileId, fileName, text, content (String, required) ▪ title (String, optional) ▪ startIndex, endIndex (Number, required)
.env Configuration	N/A
Error Handling	▪ Authorization: All routes verify that the req.userId from the JWT matches the owner of the project. A 404 Not Found is returned if the project doesn't exist or the user is not the owner, preventing data leaks.

	▪ Input Validation: Returns a 400 Bad Request for missing required fields (e.g., a code name) or malformed data (e.g., a non-array ids for bulk delete). ▪ Resource Not Found: Returns a 404 Not Found if a project or a specific sub-document (like a memo or code definition) cannot be found. ▪ Server Errors: All database operations are wrapped in try...catch blocks. Any unexpected errors result in a 500 Internal Server Error response containing error details.

1.7.5. projectExportRoutes.js

Purpose	Defines API routes to handle the exporting of various project data, such as coded segments, overlapping codes, and documents, into user-friendly file formats like Excel, DOCX, and PDF.	
Dependencies	**Internal**	../models/Project.js
	External	express exceljs docx pdfkit
Key Components	Export Routes: ▪ /export-coded-segments: A complex handler that generates highly formatted Excel reports of coded segments, supporting three different layouts ('byDocument', 'overall', 'matrix'). ▪ /export-overlaps: Calculates all instances of overlapping codes within a project and exports a detailed summary and list to an Excel file. ▪ /files/:fileId/export-memos: Exports all memos associated with a specific document to a clean Excel sheet. ▪ /files/:fileId/export: Exports the raw text content of a single document into either a .docx or .pdf file. Excel Styling Helpers: ▪ A set of internal functions (hexToArgb, createSubtleColorScheme, applySubtleColoring) dedicated to creating and applying consistent, color-coded styling to the generated Excel reports.	
Usage	All endpoints are implicitly protected and require user authentication, as they verify project ownership via req.userId.	

	GET /:projectId/export-coded-segments ▪ Description: Exports all coded segments from a project into an Excel file. ▪ URL Params: projectId (The project's ID). ▪ Query Params: format (Optional string: 'byDocument', 'overall', or 'matrix'). Defaults to 'byDocument'. ▪ Success Response: 200 OK with the generated .xlsx file. GET /:projectId/export-overlaps ▪ Description: Generates and downloads a detailed Excel report of all overlapping coded segments. ▪ URL Params: projectId (The project's ID). ▪ Success Response: 200 OK with the generated .xlsx file. GET /:projectId/files/:fileId/export-memos ▪ Description: Exports all memos from a single file within a project to an Excel file. ▪ URL Params: projectId, fileId. ▪ Success Response: 200 OK with the generated .xlsx file. GET /:projectId/files/:fileId/export ▪ Description: Exports the text content of a specific file. ▪ URL Params: projectId, fileId. ▪ Query Params: format (Required string: 'docx' or 'pdf'). ▪ Success Response: 200 OK with the generated .docx or .pdf file.
Data Schema	N/A: All endpoints use the GET method and do not process any request bodies. All necessary parameters are passed through the URL path and query string.
.env Configuration	N/A
Error Handling	▪ Authorization & Not Found: All routes first validate that the project exists and belongs to the authenticated user, returning a 404 Not Found if checks fail. Endpoints that target specific files or require data (like memos or overlaps) also return 404 if no relevant data exists to be exported. ▪ Invalid Input: The file export route (/files/:fileId/export) returns a 400 Bad Request if the format query parameter is missing or unsupported. ▪ Server Errors: Each route handler is wrapped in a try...catch block. Any failure during data processing or file generation is caught, logged to the console, and results in a 500 Internal Server Error response.

1.7.6. projectRoutes.js

Purpose	Serves as the main entry point for all project-related API routes, applying a global authentication middleware and delegating requests to specialized sub-routers.	
Dependencies	**Internal**	../controllers/projectController.js ./projectManagementRoutes.js ./projectFileManagementRoutes.js ./projectAnnotationRoutes.js ./projectExportRoutes.js
	External	express
Key Components	Global Authentication Middleware: Uses router.use(requireAuth) to protect all nested routes, ensuring every request to a project endpoint requires a valid user session. Modular Router Delegation: Consolidates multiple specialized routers (projectManagement, projectFileManagement, etc.) into a single, cohesive API structure under the /api/projects base path.	
Usage	This file is a master router and does not define individual endpoints. It delegates functionality as follows: ▪ Requests for creating, copying, or deleting entire projects are handled by projectManagementRoutes.js. ▪ Requests for uploading, committing, or managing individual files are handled by projectFileManagementRoutes.js. ▪ Requests for creating, updating, or deleting annotations (codes, memos, highlights) are handled by projectAnnotationRoutes.js. ▪ Requests for exporting project data to various formats are handled by projectExportRoutes.js.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	The primary error handling in this file is the requireAuth middleware, which will reject any request that lacks a valid authentication token with a 401	

	Unauthorized status. All subsequent error handling is delegated to the logic within the imported sub-routers.

1.7.7. statsRoutes.js

Purpose	Defines the API routes for executing statistical tests, ensuring all endpoints are protected by authentication middleware.	
Dependencies	**Internal**	../controllers/statsController.js ../controllers/authController.js
	External	express
Key Components	express.Router(): The core component used to define the /run route for statistical tests. protect Middleware: An authentication middleware that is applied to the route to ensure only logged-in users can perform statistical analyses.	
Usage	This file defines the endpoints under the /api/stats base path. POST /api/stats/run ▪ Description: Executes a statistical test. This is a protected route. The specific test and its parameters are defined in the request body. ▪ Headers: Authorization: Bearer JWT_TOKEN ▪ Success Response: 200 OK with the JSON results of the statistical test. ▪ Error Response: 401 Unauthorized if the user is not authenticated. Other errors are handled by the statsController.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	Authentication and authorization errors are handled by the protect middleware, which will return a 401 Unauthorized status. All other processing and error handling logic is delegated to the runTest controller.	

1.8. API Routes Summary

1.8.1. Authentication API (/api/auth)

Method	Endpoint	Description	Authentication Required
POST	/register	Registers a new user account.	No
POST	/login	Authenticates a user and returns a JWT.	No
POST	/forgot-password	Sends a password reset link to the user's email.	No
POST	/reset-password/:token	Sets a new password using a valid reset token.	No

1.8.2. Project Management API (/api/projects)

Method	Endpoint	Description	Authentication Required
POST	/create	Creates a new project for the authenticated user.	Yes
GET	/my-projects	Retrieves a list of all projects owned by the user.	Yes
GET	/:id	Retrieves a single project by its ID.	Yes
PUT	/:id	Updates a project's details, such as its name.	Yes
DELETE	/:id	Deletes an entire project and all its associated data.	Yes

POST	/:projectId/copy	Creates a complete duplicate of an existing project.	Yes

1.8.3. Project File Management API (/api/projects)

Method	**Endpoint**	**Description**	**Authentication Required**
POST	/:projectId/files/stage	Uploads a text file for review before saving.	Yes
POST	/:projectId/files/commit	Saves (commits) a staged file to the project database.	Yes
POST	/import-audio/:id	Uploads an audio file and replaces it with a transcript.	Yes
PUT	/:projectId/files/:fileId/rename	Renames an existing file within the project.	Yes
PUT	/:projectId/files/:fileId	Updates the text content of an existing file.	Yes
DELETE	/:projectId/files/:fileId	Deletes a file and all its associated annotations.	Yes

1.8.4. Project Annotation API (/api/projects)

Method	**Endpoint**	**Description**	**Authentication Required**
POST	/:projectId/code-definitions	Adds a new code (tag) definition to the project.	Yes

PUT	/:projectId/code-definitions/:codeDefId	Updates an existing code definition.	Yes
DELETE	/:projectId/code-definitions/:codeDefId	Deletes a code definition and its associated segments.	Yes
POST	/:projectId/codes/merge	Merges multiple source codes into one new code.	Yes
POST	/:projectId/codes/split	Splits one code into multiple new codes.	Yes
POST	/:projectId/code	Creates a new coded segment (applies a code to text).	Yes
PUT	/:projectId/code/:segmentId	Updates an existing coded segment.	Yes
DELETE	/:projectId/code/:codeId	Deletes a single coded segment.	Yes
POST	/:projectId/highlight	Adds a new inline text highlight.	Yes
DELETE	/:projectId/highlight/:highlightId	Deletes a single highlight.	Yes
POST	/:projectId/highlight/delete-bulk	Deletes multiple highlights in a single request.	Yes
POST	/:projectId/memos	Adds a new memo to a text segment.	Yes
PUT	/:projectId/memos/:memoId	Updates an existing memo.	Yes
DELETE	/:projectId/memos/:memoId	Deletes a memo.	Yes

1.8.5. Project Export API (/api/projects)

Method	Endpoint	Description	Authentication Required
GET	/:projectId/export-coded-segments	Exports coded segments into an Excel (.xlsx) file.	Yes
GET	/:projectId/export-overlaps	Exports a report of overlapping codes to Excel.	Yes
GET	/:projectId/files/:fileId/export-memos	Exports all memos from a single file to Excel.	Yes
GET	/:projectId/files/:fileId/export	Exports a file's raw content as a DOCX or PDF.	Yes

1.8.6. Statistics API (/api/stats)

Method	Endpoint	Description	Authentication Required
POST	/run	Executes a statistical test (e.g., Chi-Square).	Yes

1.9. Supporting Files and Directories

Beyond the core application logic in the src/ directory, the project root contains several standard files and folders that handle dependency management, environment configuration, testing, and deployment.

- **uploads/**: The on-disk storage location for user-uploaded files. It is subdivided into audio/ and text/ directories.
- **__tests__/**: The directory containing all automated test files for the project.
- **node_modules/**: The directory where all third-party project dependencies are installed. This folder is managed by the npm package manager and is excluded from version control.
- **.Dockerignore**: Specifies which files should be excluded from the Docker container to ensure a lean and secure build.

- **.env & .env.test**: Configuration files for storing environment variables. These files contain sensitive information such as the database connection string (MONGO_URI), server port (PORT), security secrets (JWT_SECRET), email credentials (EMAIL_USER, EMAIL_PASS), external API keys (ASSEMBLYAI_API_KEY), and application URLs (CLIENT_URL, PYTHON_API_URL). They are kept out of version control via .gitignore to protect sensitive data.
- **.gitignore**: A configuration file for the Git version control system that specifies which files and directories to ignore (e.g., node_modules/, .env).
- **Dockerfile**: A script containing instructions to build a portable Docker container for the application, which simplifies deployment.
- **jest.config.js**: The configuration file for the Jest testing framework.
- **package-lock.json**: An auto-generated file that locks the exact versions of all dependencies. This ensures that the project can be installed consistently across different machines.
- **package.json**: The project's manifest file. It lists metadata, defines dependencies required for the project, and configures shortcut scripts for running, testing, and building the application.

2. Frontend

The frontend is a modern single-page application (SPA) built with React and the Vite build tool. It is responsible for the entire user interface and user experience, providing a dynamic and responsive platform for users to interact with. It communicates with the backend via a REST API to handle user authentication, project management, data annotation, and visualization of statistical results.

2.1. frontend Directory

The project follows a component-based architecture, which is reflected in its directory structure. This organization promotes code reusability and maintainability.

```
frontend/
├── index.html
├── src/
│   ├── main.jsx
│   ├── App.jsx
│   ├── index.css
│   ├── pages/
│   │   ├── ProjectContext.jsx
│   │   ├── auth/
│   │   │   ├── AuthContext.jsx
│   │   │   ├── PrivateRoute.jsx
│   │   │   ├── Signup.jsx
│   │   │   ├── Login.jsx
│   │   │   ├── ForgotPassword.jsx
│   │   │   └── ResetPassword.jsx
│   │   ├── code/
│   │   │   ├── DefineCodeModal.jsx
│   │   │   ├── CodeDetailsModal.jsx
│   │   │   ├── FloatingAssignCode.jsx
│   │   │   ├── CodeTooltip.jsx
│   │   │   ├── SplitMergeCodesModal.jsx
│   │   │   └── SplitReviewModal.jsx
│   │   ├── components/
│   │   │   ├── ColorPicker.jsx
│   │   │   ├── ConfirmationModal.jsx
│   │   │   ├── SearchableMultiCodeDropdown.jsx
│   │   │   └── SearchableMultiSelectDropdown.jsx
│   │   ├── home/
│   │   │   ├── Home.jsx
│   │   │   ├── HomePageAnimation.jsx
│   │   │   ├── e1-logo.png
│   │   │   ├── logo.png
│   │   │   ├── TextType.jsx
│   │   │   └── TextType.css
│   │   ├── hooks/
│   │   │   ├── Hooks.jsx
```

```
│   │   │   ├── useHistory.js
│   │   │   ├── useStatsLogic.js
│   │   │   └── useTableData.js
│   │   ├── layout/
│   │   │   ├── edit-mode/
│   │   │   │   ├── EditToolbar.jsx
│   │   │   │   ├── DialogueCard.jsx
│   │   │   │   └── TranscriptEditor.jsx
│   │   │   ├── AudioPlayer.jsx
│   │   │   ├── DocumentToolbar.jsx
│   │   │   ├── DocumentViewer.jsx
│   │   │   ├── FloatingToolbar.jsx
│   │   │   ├── ImportOptionsModal.jsx
│   │   │   ├── LeftPanel.jsx
│   │   │   ├── Navbar.jsx
│   │   │   ├── PreferencesModal.jsx
│   │   │   └── ProjectView.jsx
│   │   ├── memo/
│   │   │   ├── FloatingMemoInput.jsx
│   │   │   └── MemoModal.jsx
│   │   ├── project/
│   │   │   ├── CreateProjectModal.jsx
│   │   │   ├── EditProjectModal.jsx
│   │   │   └── Projects.jsx
│   │   ├── stats/
│   │   │   ├── chi-squared/
│   │   │   │   ├── ChiSquareControlPanel.jsx
│   │   │   │   ├── ChiSquareDisplay.jsx
│   │   │   │   ├── ChiSquareDistributionChart.jsx
│   │   │   │   ├── ChiSquareTypeSelector.jsx
│   │   │   │   └── ObservedFrequencyTable.jsx
│   │   │   ├── CombineCategoriesModal.jsx
│   │   │   ├── ExpectedFrequencyDetails.jsx
│   │   │   ├── StatsModal.jsx
│   │   │   └── StatsResultsPanel.jsx
│   │   ├── table/
│   │   │   ├── ChartRenderer.jsx
│   │   │   ├── CodedSegmentsTableModal.jsx
│   │   │   ├── D3WordCloud.jsx
│   │   │   ├── StatsView.jsx
│   │   │   ├── TableView.jsx
│   │   │   └── VisualizationsView.jsx
│   │   ├── theme/
│   │   │   ├── Logo.jsx
│   │   │   ├── ThemeContext.jsx
│   │   │   └── ThemeToggle.jsx
├── public/
├── node_modules/
├── tests/
├── .Dockerignore
```

```
├── .env
├── .env.test
├── .gitignore
├── Dockerfile
├── eslint.config.js
├── tailwind.config.js
├── vite.config.js
├── package-lock.json
└── package.json
```

2.2. Commands

This section details the primary commands used to install dependencies, develop, and build the frontend application. These commands are defined in the scripts section of the package.json file and are executed using the Node Package Manager (npm).

2.2.1. npm run dev

- Purpose: To start the application in development mode with hot-reloading.
- Description: This command uses Vite to launch a local development server. It automatically monitors for file changes and instantly updates the application in the browser, providing a fast and efficient development workflow.
- Script: npm run dev

2.2.2. npm run build

- Purpose: To create a production-ready, optimized build of the application.
- Description: This command uses Vite to compile, bundle, and minify the application's source code and assets into a static dist folder. The resulting files are optimized for performance and are ready for deployment to a web server.
- Script: npm run build

2.2.3. npm run lint

- Purpose: To analyze the source code for potential errors and style inconsistencies.
- Description: This command runs ESLint across the entire codebase to identify and report on stylistic issues, potential bugs, and code that doesn't adhere to configured standards, helping to maintain code quality.
- Script: npm run lint

2.2.4. npm test

- Purpose: To execute the project's automated test suite.
- Description: This command launches Vitest, a fast and modern test runner designed to integrate seamlessly with Vite. It finds and runs all test files in the project, providing feedback on test successes and failures.
- Script: npm test

2.2.5. npm run preview

- Purpose: To locally preview the production build.
- Description: After running npm run build, this command starts a local static web server to serve the optimized files from the dist folder. This is useful for verifying that the production build works as expected before deploying it.
- Script: npm run preview

2.3. Setup

2.3.1. index.html

Purpose	Serves as the main entry point for the single-page web application, responsible for loading essential fonts and styles, setting up the root container for the React application, and running a pre-render script to establish the visual theme.	
Dependencies	**Internal**	/logo.svg /src/main.jsx /@vite/client & /@react-refresh
	External	Google Fonts (El Messiri, Julius Sans One) Boxicons CSS from a CDN (boxicons.min.css)
Key Components	Theme Initialization Script: An inline <script> located in the <head> that executes before the page renders. It checks localStorage and the user's system preferences (prefers-color-scheme) to apply the correct theme (dark or light) to the root <html> element, preventing a "flash of incorrect theme" on load. Root Element: A which acts as the mount point for the entire React application. Body Classes: The <body> tag is pre-styled with classes for background color, text color, and transitions to ensure a consistent look and feel for both light and dark modes.	
Usage	This file is the first asset served to a user's browser. The browser parses it to fetch linked stylesheets and fonts, then executes the JavaScript defined in /src/main.jsx, which in turn renders the React application inside the . During development, it is used by the Vite dev server to inject client-side code for features like live reloading.	
Data Schema	N/A	

.env Configuration	N/A
Error Handling	▪ This file has no explicit error handling. If a linked resource (CSS, font, script) fails to load, the browser's default error handling will apply, which may result in an unstyled or non-functional application.

2.4. Source Code Files

2.4.1. main.jsx

Purpose	Serves as the main entry point for the React application, rendering the root component and wrapping it with global context providers for theme and authentication.	
Dependencies	**Internal**	./index.css ./App.jsx ./pages/theme/ThemeContext.jsx ./pages/auth/AuthContext.jsx
	External	react react-dom/client
Key Components	createRoot: Initializes the React 18 concurrent rendering root, targeting the element in the main HTML file. <StrictMode>: Wraps the application to enable additional checks and warnings for potential problems during development. <ThemeProvider>: A context provider that makes theme-related state and functions (e.g., toggling dark/light mode) available to the entire component tree. <AuthProvider>: A context provider that manages and provides user authentication state (e.g., user data, token) throughout the application.	
Usage	This is the client-side entry point for the React application, executed by the browser to start the program. It is not imported by other modules.	
Data Schema	N/A	

.env Configuration	N/A
Error Handling	▪ This file uses React's <StrictMode> to help detect potential problems and bugs during development. Runtime application errors are handled within the React component tree, not at this entry point.

2.4.2. App.jsx

Purpose	Serves as the application's root component, defining the entire routing structure with react-router-dom and wrapping all pages in a global ProjectContext.	
Dependencies	**Internal**	./index.css ./pages/auth/Signup.jsx ./pages/auth/Login.jsx ./pages/auth/ForgotPassword.jsx ./pages/auth/ResetPassword.jsx ./pages/auth/PrivateRoute.jsx ./pages/home/Home.jsx ./pages/project/Projects.jsx ./pages/layout/ProjectView.jsx ./pages/ProjectContext.jsx ./pages/theme/ThemeToggle.jsx
	External	react-router-dom
Key Components	<BrowserRouter>: Enables client-side routing, allowing the application to navigate between pages without full page reloads. <Routes>: The main container for all individual route definitions. <Route>: Maps a specific URL path to a React component (e.g., the path /login renders the Login component). <PrivateRoute>: A custom wrapper component that protects specific routes by requiring user authentication before rendering the intended page.	

	<ThemeToggle />: A globally available component that allows users to switch between light and dark themes.
Usage	This component is the top-level container for the entire application, rendered directly by main.jsx. It establishes all navigable pages and ensures that components like the theme toggle are present across the site.
Data Schema	N/A
.env Configuration	N/A
Error Handling	▪ Routing: react-router-dom handles unmatched URLs by default (typically rendering a blank page). A custom 404-Not-Found route could be added for a better user experience. ▪ Authorization: The <PrivateRoute> component is responsible for handling authorization errors by preventing unauthenticated users from accessing protected pages, likely redirecting them to the login page.

2.4.3. index.css

Purpose	Defines global CSS styles and custom, reusable utility classes for the application using Tailwind CSS.	
Dependencies	**Internal**	N/A
	External	tailwindcss
Key Components	.custom-scrollbar: A utility that applies a consistent, minimal scrollbar style for WebKit browsers, with distinct appearances for light and dark themes. .hide-number-arrows: A class to remove the default spinner arrows from HTML number input fields. .el-messiri-bold: A typography utility that applies the "El Messiri" font with a bold weight. .julius-sans-one-regular: A typography utility that applies the "Julius Sans One" font with a regular weight.	
Usage	This file is imported globally (typically in main.jsx) to make its styles available throughout the entire application. Developers can use the defined classes like .custom-scrollbar and .el-messiri-bold in any component's className prop to apply these specific styles.	

Data Schema	N/A
.env Configuration	N/A
Error Handling	▪ CSS does not have runtime error handling. Syntax errors are reported in the browser's developer console, which may cause styles to not apply correctly. Misspelling a class name in a component will result in the style simply not being applied, without throwing an error.

2.4.4. ProjectContext.jsx

Purpose	Creates a centralized React Context to share project-related state and functions throughout the application, preventing the need to pass props down through many component levels.	
Dependencies	**Internal**	N/A
	External	react
Key Components	ProjectContext: The exported React Context object created by createContext(). Any component can subscribe to it to access shared project data.	
Usage	This file exports a Context object, not a component. A developer uses it in two primary ways: ▪ Wrap a part of the component tree with <ProjectContext.Provider value={...}> to make project data available. ▪ In any child component, use the useContext(ProjectContext) hook to access that data.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	▪ This file contains no runtime logic. Errors typically occur if useContext(ProjectContext) is called in a component that is not a descendant of a <ProjectContext.Provider>, which React will report during development.	

2.4.5. auth

2.4.5.1. AuthContext.jsx

Purpose	Provides a centralized system for managing and sharing user authentication state across the application, handling session persistence, login/logout logic, and the current user's data.	
Dependencies	**Internal**	N/A
	External	react
Key Components	AuthProvider: A provider component that wraps parts of the application. It manages the user state, persists the session to localStorage, and exposes authentication functions (login, logout) through context. useAuth: A custom hook that simplifies access to the authentication context, allowing any child component to easily retrieve the user's status and authentication functions.	
Usage	A developer should wrap the application's root component (or any part needing authentication) with the <AuthProvider> in main.jsx. Then, within any child component, call the useAuth() hook to access the authentication state and methods.	
Data Schema	The useAuth hook returns an object with the following structure: ▪ user (Object | null): Contains the logged-in user's data, or null if logged out. ▪ isAuthenticated (Boolean): A flag that is true if a user with a token is logged in. ▪ loading (Boolean): true on initial load while checking for a persisted session, otherwise false. ▪ login(userData) (Function): Sets the user state and saves the session to localStorage. ▪ logout() (Function): Clears the user state and removes the session from localStorage.	
.env Configuration	N/A	
Error Handling	▪ The provider includes a try...catch block to safely parse user data from localStorage. If the stored data is corrupted, it will be cleared, and the user will be logged out to prevent application crashes.	

2.4.5.2. PrivateRoute.jsx

Purpose	Acts as a component-based route guard to protect parts of the application that require a user to be logged in, redirecting unauthenticated users to the login page.	
Dependencies	**Internal**	./AuthContext
	External	react-router-dom
Key Components	PrivateRoute: The main wrapper component that contains the core logic. Conditional Rendering: The component's logic checks the authentication status and returns one of three things: ▪ A loading indicator if the authentication status is still being determined. ▪ A <Navigate> component to redirect to /login if the user is not authenticated. ▪ The children prop (the protected component) if the user is authenticated.	
Usage	This component is used in the main router configuration (App.jsx) to wrap routes that should be protected. Any component passed as children to PrivateRoute will only be rendered if the user is authenticated.	
Data Schema	This component passes a state object to the /login route upon redirection: ▪ { from: location }: This object contains the original URL the user was trying to access. The Login component can use this to redirect the user back to their intended page after a successful login.	
.env Configuration	N/A	
Error Handling	▪ This component is designed to handle the "not authenticated" access case by redirecting the user. It also gracefully manages the initial authentication check by displaying a loading state, preventing a premature redirect before the user's session status is confirmed.	

2.4.5.3. Signup.jsx

Purpose	Renders the user registration page, providing a form for new users to create an account and handling the API request to the backend.	
Dependencies	**Internal**	./AuthContext

	External	react axios react-router-dom
Key Components	State Management: Uses the useState hook to manage form data, API error messages, and the loading status of the submission button. handleChange(e): Updates the component's state as the user types in the name, email, and password fields. handleSubmit(e): An asynchronous function that sends the registration data to the backend API. Upon success, it logs the user in using the AuthContext and navigates them to the /projects page.	
Usage	This component is a page, intended to be rendered by the main application router (App.jsx) when the user navigates to the /signup URL.	
Data Schema	This component sends a POST request to the registration endpoint with the following body structure: Registration Body: ▪ name (String, required) ▪ email (String, required) ▪ password (String, required)	
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoint for the registration request.	
Error Handling	▪ The handleSubmit function uses a try...catch block to handle API errors. If the backend returns an error (e.g., "User already exists"), the message is extracted from the response and displayed to the user. ▪ A loading state is used to disable the submit button during the API call, preventing duplicate submissions.	

2.4.5.4. Login.jsx

Purpose	Renders the user login page, which provides a form to capture user credentials, sends them to the backend for verification, and manages the session on successful authentication.	
Dependencies	**Internal**	./AuthContext
	External	react

		axios react-router-dom
Key Components	State Management: Uses the useState hook to manage the form's input data, API error messages, and the loading state of the submit button. handleChange(e): Updates the component's state as the user types their email and password. handleSubmit(e): An asynchronous function that sends the user's credentials to the backend API. On success, it updates the global authentication state via AuthContext and navigates the user to their projects page.	
Usage	This component is a page rendered by the main application router (App.jsx) when the user navigates to the /login URL. It is also the redirect target for unauthenticated users attempting to access protected routes.	
Data Schema	This component sends a POST request to the login endpoint with the following body structure: Login Body: ▪ email (String, required) ▪ password (String, required)	
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoint for the login request.	
Error Handling	▪ The handleSubmit function uses a try...catch block to handle API errors. If the backend returns an error (e.g., "Invalid Credentials"), the message is extracted from the response and displayed to the user. ▪ A loading state is used to disable the submit button during the API call, preventing multiple submissions.	

2.4.5.5. ForgotPassword.jsx

Purpose	Renders a page with a form that allows a user to request a password reset link by submitting their email address.	
Dependencies	**Internal**	N/A
	External	react axios

Key Components	State Management: Uses the useState hook to manage the email input field, feedback messages (for success or error), and the loading state. handleSubmit(e): An asynchronous function that sends the user's email to the backend API to initiate the password reset process. User Feedback: The component displays a success or error message to the user based on the outcome of the API request.
Usage	This is a page component rendered by the main application router (App.jsx) when the user navigates to the /forgot-password URL.
Data Schema	This component sends a POST request to the forgot-password endpoint with the following body structure: Forgot Password Body: ▪ email (String, required)
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoint for the password reset request.
Error Handling	▪ The handleSubmit function uses a try...catch block to handle API errors. If the request fails, an error message is displayed to the user. ▪ A loading state is used to disable the form during the API call, preventing multiple submissions. ▪ The success message is intentionally generic (e.g., "Check your email...") to avoid confirming whether an email address is registered in the system.

2.4.5.6. ResetPassword.jsx

Purpose	Renders a page with a form that allows a user to set a new password, using a unique token from the URL to authorize the change.	
Dependencies	**Internal**	N/A
	External	react axios react-router-dom
Key Components	State Management: Uses the useState hook to manage the new password and confirmation fields, as well as error and message states for providing user feedback.	

	handleSubmit(e): An asynchronous function that validates the form on submission. It ensures the passwords match before sending a POST request to the backend with the token and new password. API Interaction: Uses useParams to extract the reset token from the URL and axios to communicate with the password reset endpoint.
Usage	This is a page component rendered by the main application router (App.jsx) when a user navigates to a URL matching the /reset-password/:token pattern. Users typically access this page by clicking a unique link sent to their email.
Data Schema	This component sends a POST request to the reset password endpoint. The token is sent as a URL parameter. Reset Password Body: ▪ password (String, required)
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoint for the password reset request.
Error Handling	▪ Client-side Validation: Before submitting, the component checks if the "password" and "confirm password" fields match and displays an error if they don't. ▪ API Errors: The handleSubmit function uses a try...catch block to handle API errors. If the backend returns an error (e.g., "Invalid or expired token"), the message is extracted from the response and displayed to the user.

2.4.6. code

2.4.6.1. DefineCodeModal.jsx

Purpose	Provides a reusable modal form for both creating new code definitions and editing existing ones, complete with client-side validation and state management.	
Dependencies	**Internal**	../components/ColorPicker
	External	react framer-motion
Key Components	Conditional Modes: The modal operates in either "create" or "edit" mode, determined by the presence of the initialCode prop. This changes the UI text and pre-populates the form for editing.	

	useEffect Hook: Initializes and resets the form's state whenever the modal is opened, ensuring clean data for both creating and editing tasks. useMemo Hook: Optimizes performance by memoizing the list of colors already in use, which is used for client-side validation to prevent duplicate color selection. handleSubmit(e): A handler that performs client-side validation (e.g., for empty name, duplicate color) before passing the final data to the parent component via the onSave callback.
Usage	This is a controlled component. A parent component must manage the show state to control its visibility. To create a new code, pass an onSave callback. To edit an existing code, pass both the onSave callback and the initialCode object.
Data Schema	This component does not make API calls. It passes the following data object to the parent component through the onSave callback: ▪ onSave Payload: { name, description, color, _id } (_id is included when editing).
.env Configuration	N/A
Error Handling	▪ Client-side Validation: The component checks for empty code names and duplicate color usage before submission, displaying an error message if validation fails. ▪ Backend Error Display: It includes a mechanism to display validation errors that originate from the backend (e.g., duplicate name), which are passed down from the parent component. ▪ Dismissal: The modal can be closed by clicking the backdrop, a "Cancel" button, or an "X" icon, all of which trigger the onClose callback.

2.4.6.2. CodeDetailsModal.jsx

Purpose	Displays a modal window with detailed information about either a code definition or a single coded segment, including statistical data like frequency counts.	
Dependencies	**Internal**	N/A
	External	react

		framer-motion react-icons
Key Components	Conditional Rendering: The modal dynamically changes its content to show either details for a Code Definition (including its description and frequency) or a Coded Segment (including its text and location) based on the props it receives. useMemo for Calculations: Efficiently calculates the global (project-wide) and local (current file) frequency of a code. This calculation is memoized to prevent unnecessary re-computation on re-renders, optimizing performance. framer-motion Integration: Uses <AnimatePresence> and <motion.div> to provide smooth fade and scale animations when the modal enters and exits the screen.	
Usage	This component is controlled by a parent. A developer should manage state in the parent to control the show prop and pass the relevant data object (codeDefinition or codeSegment), a list of all segments for calculations, and an onClose function to handle its dismissal.	
Data Schema	This component expects specific data structures via its props: ▪ codeDefinition Prop: An object containing _id, name, description, and color. ▪ codeSegment Prop: An object containing text, fileName, startIndex, endIndex, and a nested codeDefinition object. ▪ allCodedSegments Prop: An array of all coded segment objects from the project.	
.env Configuration	N/A	
Error Handling	▪ The component gracefully handles optional or missing data (e.g., displaying "No description provided." if a code has no description). ▪ It uses e.stopPropagation() on the modal content to prevent accidental closing when a user clicks inside the modal itself.	

2.4.6.3. FloatingAssignCode.jsx

Purpose	Renders a floating toolbar at specific screen coordinates to allow a user to quickly assign a code to a text selection from a searchable list.	
Dependencies	**Internal**	N/A

	External	react framer-motion react-icons
Key Components	Absolute Positioning: The component uses x and y props to position itself as a fixed element on the screen, typically next to a user's text selection. Search and Filter: An input field allows the user to filter the list of available codes in real-time by searching their name and description. Callback-driven Actions: The component is entirely controlled by props. It uses callback functions (onAssignCode, onDefineNewCode, onClose) to communicate user actions back to its parent component. framer-motion Animations: Utilizes <AnimatePresence> and <motion.div> to provide smooth entrance and exit animations.	
Usage	This is a controlled component. A developer should render it from a parent component that tracks text selections. The parent is responsible for calculating the x and y coordinates, controlling visibility, and providing the list of codeDefinitions and all necessary callback functions.	
Data Schema	This component expects props with the following data structures: ▪ codeDefinitions Prop: An array of objects, where each object must have an _id, name, and color. ▪ onAssignCode Callback: This function is invoked with a single argument: the _id of the code the user selected.	
.env Configuration	N/A	
Error Handling	▪ The component relies on the parent to handle its visibility and dismissal via the onClose callback. ▪ It uses e.stopPropagation() on its root element to prevent clicks within the component from unintentionally triggering events in the underlying page content.	

2.4.6.4. CodeTooltip.jsx

Purpose	Renders a styled tooltip in a fixed position on the screen to display the name and color of one or more codes, typically in response to a user hovering over a coded segment.

Dependencies	**Internal**	N/A
	External	react framer-motion
Key Components	Conditional Rendering: The component only renders if its visible prop is true and it has received an array of codes. The tooltip's title also dynamically changes from "Code" to "Overlapping Codes" if multiple codes are displayed. framer-motion Integration: Uses <AnimatePresence> and <motion.div> to provide a smooth slide-and-fade animation when the tooltip appears and disappears. Fixed Positioning: The component is styled to always appear in the bottom-right corner of the viewport.	
Usage	This is a controlled component. A parent component should manage the visible state and the codes array, passing them as props to show or hide the tooltip based on user actions like hovering over an element.	
Data Schema	The component expects its codes prop to be an array of objects, where each object has the following structure: ▪ { _id: '...', codeDefinition: { name: '...', color: '...' } }	
.env Configuration	N/A	
Error Handling	▪ The component gracefully handles potentially missing data by providing fallback values (e.g., 'Unnamed Code') if a code's properties are not defined. ▪ It will not render if the codes array is empty, preventing an empty tooltip from appearing.	

2.4.6.5. SplitMergeCodesModal.jsx

Purpose	Provides a modal for advanced codebook management, featuring a tabbed interface for two distinct operations: merging multiple codes into a single new code, or splitting a single code into several new ones.	
Dependencies	**Internal**	../components/ColorPicker
	External	react framer-motion

		react-icons
Key Components	Tabbed Interface: A state-driven UI that allows the user to switch between the "Merge Codes" and "Split Code" functionalities within a single modal. Dynamic Forms: Both the merge and split forms allow users to dynamically add or remove input fields, enabling the merging of many source codes or splitting into many new codes. Client-side Validation: Includes robust validation logic to check for common errors before submission, such as requiring a minimum number of codes for a merge, ensuring unique names and colors for new codes, and preventing empty submissions. findAvailableColor Helper: An intelligent utility function that automatically suggests an unused color for newly defined codes, prioritizing a standard color palette before generating a random color. useEffect and useMemo Hooks: Used extensively to reset the modal's state when opened, manage UI side effects like closing popovers, and optimize performance by memoizing derived data like the set of used colors.	
Usage	This is a controlled component. A parent component must manage the show prop to control its visibility and provide the necessary data and callbacks: onClose, codeDefinitions, allCodedSegments, onMerge, and onInitiateSplit.	
Data Schema	This component communicates with its parent via callback functions, passing the following data structures: ▪ onMerge Payload: An object { sourceCodeIds: [...], newCodeName: '...', newCodeColor: '...' }. ▪ onInitiateSplit Payload: A function called with two arguments: (sourceCodeId, newCodeDefinitions), where newCodeDefinitions is an array of { name, color } objects.	
.env Configuration	N/A	
Error Handling	▪ User Feedback: Displays clear, animated error messages within the modal for validation failures. These messages automatically disappear after a few seconds. ▪ Input Validation: Prevents form submission and informs the user if required fields are empty, if selections are not unique (e.g., merging the same code with itself), or if new code names/colors conflict with each other or existing codes.	

	▪ Interaction Handling: Manages clicks outside of popovers to close them, preventing UI clutter.

2.4.6.6. SplitReviewModal.jsx

Purpose	Guides a user through the mandatory re-categorization of coded segments after a "split code" operation, presenting each segment one-by-one for assignment to a new code.	
Dependencies	**Internal**	N/A
	External	react react-icons framer-motion
Key Components	Step-by-Step Review UI: The component functions as a wizard, displaying one segment at a time along with buttons for each of the newly defined codes, plus an "Un-code" option. State Management: Uses useState to track the currentIndex of the segment being reviewed and an assignments object to store the user's choices. handleAssign(newCodeName): The core logic handler. It records the user's assignment for the current segment and advances to the next. When the last segment is reviewed, it triggers the onCompleteSplit callback with all the assignments. Progress Indicator: A visual progress bar and text (Segment X of Y) clearly communicate the user's progress through the review process.	
Usage	This is a controlled component intended to be shown immediately after a user defines the new codes in the SplitMergeCodesModal. The parent component must provide the show prop, the original sourceCode, the array of segmentsToReview, the array of newCodes, and the onCompleteSplit and onClose callback functions.	
Data Schema	This component communicates the final results to its parent via the onCompleteSplit callback. ▪ onCompleteSplit Payload: An assignments object where keys are the original segment IDs and values are the names of the new codes they have been assigned to (or null if un-coded). ▪ Example: { "segmentId1": "New Code A", "segmentId2": null }	

.env Configuration	N/A
Error Handling	▪ Cancellation: The user can cancel the entire process at any time by closing the modal, which calls the onClose function. ▪ Navigation: A "Previous" button allows the user to go back and correct a previous assignment before finalizing the split. ▪ The component's design ensures that the onCompleteSplit function is only called after every single segment has been reviewed, preventing an incomplete operation.

2.4.7. components

2.4.7.1. ColorPicker.jsx

Purpose	Provides a reusable UI component for selecting colors, offering a predefined palette, a custom color option via the native browser picker, and the ability to disable unavailable colors.	
Dependencies	**Internal**	N/A
	External	react
Key Components	Standard Color Palette: Displays a predefined set of selectable color swatches. Custom Color Picker: Features a custom-styled button that programmatically triggers the browser's native <input type="color"> element, allowing for a seamless user experience with a custom UI. Disabled State: Can receive an array of usedColors, which it then visually disables in the palette, preventing users from selecting them. useMemo Hook: Optimizes the process of checking for used colors by converting the usedColors array into a Set for faster lookups.	
Usage	This is a controlled component. A developer should use it within a form by providing the current color value from a parent's state and an onChange function to handle updates when a new color is selected.	
Data Schema	The component's interface is defined by its props: ▪ color (String, required): The hex string of the currently selected color. ▪ onChange (Function, required): A callback function that receives the new hex color string when the selection changes. ▪ usedColors (Array of String, optional): An array of hex color strings to be disabled.	

.env Configuration	N/A
Error Handling	▪ The component provides clear visual cues for its state: the currently selected color is highlighted, and unavailable colors are grayed out with a strikethrough. ▪ It relies on the parent component to manage and validate the color state.

2.4.7.2. ConfirmationModal.jsx

Purpose	Provides a highly configurable and reusable modal component for handling various user confirmation scenarios, ranging from simple dialogs to critical actions that require additional validation.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	Configurability via Props: The component's appearance and behavior are controlled almost entirely through props, allowing it to be adapted for different scenarios (e.g., showing a text input for destructive actions with showInput, or a required checkbox with showCheckbox). Conditional Confirmation Logic: The main confirmation button is intelligently disabled based on the configuration. It remains disabled until the user meets the required conditions, such as typing a specific phrase or checking a required box. State Management: It internally manages the state for user inputs like the text field and checkbox, and resets this state each time the modal is opened. framer-motion Integration: Uses <AnimatePresence> and <motion.div> to create smooth animations for the modal's appearance and for revealing detailed messages.	
Usage	This is a controlled component. A developer must manage the show state in a parent component to control its visibility. The modal's content and functionality are customized by passing different props. The onClose and onConfirm callbacks are essential for handling user actions.	

Data Schema	This component's data interface is its props. The onConfirm callback function receives one argument: ▪ isChecked (Boolean): The state of the checkbox at the time of confirmation, which can be used by the parent's confirmation logic.
.env Configuration	N/A
Error Handling	▪ Prop Fallbacks: The component defines defaultProps to ensure that onClose and onConfirm are always functions, preventing crashes if they are not provided. ▪ User Input Validation: Prevents confirmation until required actions are completed by the user (e.g., typing a confirmation phrase), providing a safe user experience for critical operations. ▪ Dismissal: The modal can be safely closed by clicking the backdrop or a "Cancel" button, which triggers the onClose callback.

2.4.7.3. SearchableMultiCodeDropdown.jsx

Purpose	Provides a reusable dropdown component for selecting multiple "codes" from a list, featuring a search filter and a context-aware "Select All" function.	
Dependencies	**Internal**	N/A
	External	react react-icons
Key Components	Search and Filter: An input field allows users to filter the list of codes in real-time, making it easy to find specific items in a large list. Context-Aware "Select All": A "Select All" checkbox that intelligently selects or deselects only the items that are currently visible in the filtered list, not the entire list of codes. Click-Outside-to-Close: An effect hook that automatically closes the dropdown menu if the user clicks anywhere outside of its boundaries, providing an intuitive user experience. Controlled Component: The selection state is managed by the parent component, making it a flexible and predictable component to integrate.	
Usage	This is a controlled component. A developer must manage the selectedCodeIds array in a parent component's state and pass it as a prop. An onSelectionChange	

	callback function must also be provided to receive the updated array of selected IDs whenever the user makes a change.
Data Schema	The component's interface is defined by its props: ▪ codes Prop: An array of objects, where each object has an _id, name, and color. ▪ selectedCodeIds Prop: An array of strings representing the _ids of the selected codes. ▪ onSelectionChange Callback: A function that receives the new array of selected _ids.
.env Configuration	N/A
Error Handling	▪ The component displays a "No codes found" message if the user's search term doesn't match any available codes. ▪ The useEffect hook for closing the dropdown on outside clicks prevents it from remaining stuck in an open state.

2.4.7.4. SearchableMultiSelectDropdown.jsx

Purpose	Provides a reusable dropdown for selecting multiple files, featuring a search filter, a "Select All" function, and the ability to disable specific files from selection.	
Dependencies	**Internal**	N/A
	External	react react-icons
Key Components	Search and Filter: An input field allows users to filter the list of files by name in real-time. Disabled Items: The component can accept an array of disabledFileIds, which are then visually grayed out and made unselectable in the list. Context-Aware "Select All": A "Select All" function that smartly applies only to the files that are currently visible in the filtered list and are not disabled. Click-Outside-to-Close: An effect hook that closes the dropdown menu when the user clicks anywhere outside of the component. File Type Icons: Displays distinct icons for audio and text files, providing a helpful visual cue.	

Usage	This is a controlled component. The parent must manage the selectedFileIds state and provide it as a prop, along with an onSelectionChange callback. The parent also supplies the complete list of files and can optionally pass an array of disabledFileIds.
Data Schema	The component's interface is defined by its props: ▪ files Prop: An array of objects, each with _id, name, and sourceType. ▪ selectedFileIds Prop: An array of strings representing the _ids of selected files. ▪ onSelectionChange Callback: A function that receives the updated array of selected _ids. ▪ disabledFileIds Prop: An optional array of file _ids to disable.
.env Configuration	N/A
Error Handling	▪ The component displays a "No files found" message if a search returns no results. ▪ It visually indicates and prevents interaction with any files passed in the disabledFileIds prop. ▪ The click-outside logic ensures a robust user experience by preventing the dropdown from being stuck open.

2.4.8. home

2.4.8.1. Home.jsx

Purpose	Renders the application's main landing page, designed to introduce the product's features and provide a clear call-to-action for both new and returning users through an engaging, animated interface.	
Dependencies	**Internal**	../auth/AuthContext ../theme/Logo ./HomePageAnimation ./TextType.jsx ./TextType.css Local image assets (.png)
	External	react

		react-router-dom framer-motion react-icons
Key Components	Two-Section Layout: The page is divided into a "hero" section (top of the page) and a "features" section, with custom logic to navigate between them. Scroll-Snapping Effect: A useEffect hook intercepts mouse wheel events to create a "snap-scrolling" behavior, smoothly transitioning the user between the hero and features sections as a single scroll action. Scroll-Triggered Animations: Uses the useInView hook from framer-motion to animate the feature cards into view only when the user scrolls down to that section. Component Imports: ▪ HomePageAnimation: Renders a complex visual animation in the hero section. ▪ TextType: Creates an animated "typing" effect for the main headline. Scroll Indicator: A floating arrow icon that provides a visual cue and click-to-scroll functionality to navigate between the two main sections.	
Usage	This component is a page rendered by the main application router (App.jsx) when the user visits the root (/) or /home path. It serves as the application's primary landing page.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	▪ This component does not feature explicit error handling. Its primary complexity lies in its animations and custom scroll logic, not in data fetching or user input that would typically require error management.	

2.4.8.2. HomePageAnimation.jsx

Purpose	Renders a complex, looping, multi-scene animation that visually demonstrates the core user workflow of the application, serving as a primary visual element for the landing page.	
Dependencies	**Internal**	../theme/Logo.jsx

	External	react framer-motion react-icons recharts
Key Components	Animation State Machine: The component operates as a state machine, transitioning through a predefined sequence of "scenes" (e.g., IMPORT, WORKSPACE, VISUALIZATION) to showcase different application features. Simulated User Interaction: It creates an animated cursor that simulates user actions like clicking buttons, selecting text, and applying codes, providing a dynamic product demonstration. useEffect Animation Loop: A master useEffect hook orchestrates the entire animation sequence, including the timing of each scene and the detailed steps within them. The animation automatically loops upon completion. Lifecycle Management: Includes robust cleanup logic using useRef and a visibilitychange event listener to pause or abort the animation if the component unmounts or the browser tab is hidden, preventing errors and conserving resources. Child Components: Composed of several static child components like StaticAudioPlayer and StaticChiSquareDistributionChart to represent different parts of the UI during the animation.	
Usage	This is a self-contained presentational component. A developer should import it and place it directly into the home page layout. It requires no props to function.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	▪ The component is designed for resilience. It includes robust lifecycle management to automatically cancel animations and prevent state updates if the component is unmounted or the browser tab becomes inactive, which is its primary form of error prevention.	

2.4.8.3. TextType.jsx

Purpose	Provides a versatile and highly customizable React component for creating animated "typing" effects, suitable for dynamic headlines and presentations.	
Dependencies	**Internal**	./TextType.css
	External	react gsap
Key Components	Extensive Configurability: The component's behavior is controlled by a rich set of props, allowing customization of typing/deleting speeds, looping, cursor appearance, and more. useEffect Animation Engine: A complex useEffect hook orchestrates the entire animation cycle, using setTimeout to handle character-by-character typing, pausing, and deleting logic. GSAP for Cursor Animation: Utilizes the GSAP library to create a smooth and performant blinking animation for the cursor. Intersection Observer: Features a startOnVisible prop that uses the IntersectionObserver API to delay the animation until the component scrolls into the viewport, optimizing page performance. Dynamic Element Rendering: Uses React.createElement to render its container as any specified HTML tag or React component (via the as prop), making it semantically flexible.	
Usage	This is a presentational component. A developer imports it and configures its animation by passing various props. It can be used for a single sentence, or an array of sentences for a more complex, looping effect. Callbacks like onComplete can be used to trigger other events.	
Data Schema	The component's interface is defined by its props. Key props include: ▪ text (String or Array of String, required): The content to be typed. ▪ typingSpeed, deletingSpeed, pauseDuration (Number): Control the timing of the animation. ▪ loop (Boolean): Determines if the animation repeats. ▪ as (String or React Element): The container element to render. ▪ onComplete (Function): A callback fired when a non-looping animation finishes.	
.env Configuration	N/A	

Error Handling	▪ Callback Safety: The onSentenceComplete and onComplete callbacks are wrapped in try...catch blocks, preventing errors within the parent's callback from crashing the animation. ▪ Cleanup: The main animation useEffect hook returns a cleanup function that clears any pending timeouts, preventing memory leaks when the component unmounts or re-renders.

2.4.8.4. TextType.css

Purpose	Provides the essential CSS styles for the TextType.jsx component, ensuring correct layout for the text and proper styling and visibility for the animated cursor.	
Dependencies	**Internal**	N/A
	External	N/A
Key Components	.text-type: The main container class that ensures line breaks and spacing in the typed text are preserved. .text-type__cursor: Styles the cursor element, setting its position and default appearance. .text-type__cursor--hidden: A utility class used by the JavaScript component to hide the cursor during the typing animation when configured to do so.	
Usage	This stylesheet should be imported into the TextType.jsx component or a global CSS file. The classes are automatically applied by the TextType.jsx component and do not need to be manually used by a developer.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	▪ This is a stylesheet and does not have runtime error handling. If the file fails to load, the TextType component will render without its intended styling.	

2.4.9. hooks

2.4.9.1. Hooks.jsx

Purpose	Encapsulates the entire state and business logic for the main project view, centralizing data fetching, annotation management, user interactions, modal states, search, and undo/redo history into a single, cohesive unit.	
Dependencies	**Internal**	../auth/AuthContext.jsx ./useHistory.js
	External	react react-router-dom axios file-saver
Key Components	Centralized State Management: Manages over 30 useState variables that control every aspect of the project view, from the core project data and selected file to the visibility of every modal, panel, and floating toolbar. API Logic: Contains all async functions for communicating with the backend API, including handlers for fetching the project and performing CRUD (Create, Read, Update, Delete) operations on files, code definitions, memos, and other annotations. Undo/Redo Integration: Integrates the useHistory custom hook to wrap most annotation-related actions (creating/deleting codes, memos, etc.), providing robust undo and redo capabilities. Text Selection Engine: Includes a set of utility functions (getSelectionInfo, createRangeFromOffsets) that interact with the browser's Selection API to precisely calculate the character offsets of user-selected text, which is crucial for creating annotations. Event Orchestration: A master handleViewerMouseUp event handler processes user text selections and determines which contextual floating toolbar or action to trigger. Memoization: Uses the useMemo hook to efficiently process and group annotation data for display (e.g., grouping coded segments by code name), optimizing rendering performance.	
Usage	This custom hook is not a component. It is designed to be called once at the top level of the main ProjectView component. The large object it returns contains all the state values and callback functions needed by the UI. These are	

	then passed down as props to the various child components (side panels, viewer, modals), effectively separating all complex logic from the presentational components.
Data Schema	This hook serves as the primary frontend interface to the backend API. It handles the data schemas for all CRUD operations related to a project, including those for: ▪ Files (text and audio) ▪ Code Definitions ▪ Coded Segments ▪ Inline Highlights ▪ Memos
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, which is used in all axios requests made by the hook.
Error Handling	▪ API Communication: Nearly all functions that interact with the backend are wrapped in try...catch blocks. On failure, they set an error state and often trigger a confirmation modal to display a user-friendly error message. ▪ Authentication: Checks for a user authentication token before making API calls. ▪ User Input: Handles edge cases like duplicate file names on import by showing a confirmation modal that explains the risks and suggests an alternative name.

2.4.9.2. useHistory.js

Purpose	Provides a custom React hook that implements a file-scoped undo/redo history system, allowing user actions to be reversed and re-applied on a per-document basis.	
Dependencies	**Internal**	N/A
	External	react
Key Components	State Management: Manages a single state object where keys are file IDs. Each file has its own undoStack and redoStack, ensuring that the history is specific to the document being edited. executeAction(action): The primary function for performing a new action. It runs the action, and if successful, pushes the corresponding inverse (undo) action onto the history stack and clears the redo stack.	

	undo(): Pops the last action from the undo stack, executes it to reverse the change, and moves the original action to the redo stack. redo(): Pops the last undone action from the redo stack, re-executes it, and moves it back to the undo stack. canUndo & canRedo: Derived boolean values that indicate whether there are actions available in the undo or redo stacks for the current file, useful for enabling/disabling UI buttons.
Usage	This hook is intended to be used by a parent hook or component that manages user actions. A developer wraps their data-mutating logic (e.g., API calls) into an action object with execute and undo methods, and then passes this object to the executeAction function.s
Data Schema	The hook's interface is based on the command pattern. action Object: The primary input for executeAction, which must have: ▪ execute: An async function to perform the action. ▪ undo: An object with its own execute method to perform the inverse action. Returned Object: The hook returns an object containing the executeAction, undo, and redo functions, along with the canUndo and canRedo booleans.
.env Configuration	N/A
Error Handling	▪ The hook is file-scoped and will not perform any operations if no currentFileId is provided. ▪ It only adds an action to the undo stack if its execute method returns a result object with success: true, preventing failed operations from corrupting the history.

2.4.9.3. useStatsLogic.js

Purpose	Encapsulates all state, effects, and API interactions for the Statistical Analysis feature, managing the complex multi-step workflow for configuring, validating, and executing statistical tests.	
Dependencies	**Internal**	../auth/AuthContext.jsx
	External	react axios

Key Components	State Machine: Manages the UI flow using a view state, guiding the user through different screens like test selection, configuration, validation, and results. API Interaction: Contains async functions (performValidation, executeTestApi) to communicate with the backend stats API, both for checking statistical assumptions and for running the final test calculations. Payload Generation: A getChiSquarePayload helper function dynamically constructs the correct JSON payload for the API based on the selected test type and user inputs. Client-side Validation: A memoized value, areChiSquareInputsIncomplete, performs real-time validation on the user's selections (e.g., ensuring enough codes/documents are chosen), disabling UI controls until the requirements are met. State Management: Centralizes all user input state for each Chi-Square test type (gofCodes, indepDocs, homoDocGroups, etc.) and the state for API responses (loading, error, results).
Usage	This hook is designed to be called once within the parent StatsView component. The large object it returns contains all the state and handler functions necessary to power the entire statistical analysis interface, which are then passed as props to various child components (forms, modals, charts, etc.).
Data Schema	This hook is the primary client-side interface for the /api/stats/run endpoint. It constructs and sends JSON payloads whose structure depends on the selected chiSquareSubtype, including fields like codes, docList, indepDocs, and homoDocGroups.
.env Configuration	This hook does not directly use environment variables, but the axios calls to the backend API (e.g., /api/stats/run) depend on a globally configured base URL, typically set via VITE_BACKEND_URL in a Vite project setup.
Error Handling	▪ API Errors: All API requests are wrapped in try...catch blocks. If a request fails, the error message from the backend response is captured and stored in the error state to be displayed in the UI. ▪ User Workflow: It uses confirmation modals to ensure the user acknowledges important statistical assumptions before proceeding, which helps prevent misinterpretation of results. ▪ State Reset: An useEffect hook automatically resets the validation and results state whenever the user changes any input parameters, ensuring stale data is not shown.

2.4.9.4. useTableData.js

Purpose	Encapsulates all client-side data processing for table views by filtering, sorting, and aggregating raw project data into memoized, ready-to-render data structures for different display formats.	
Dependencies	**Internal**	N/A
	External	react
Key Components	Memoized Data Views (useMemo): The core of the hook is its extensive use of useMemo to create several derived views of the project data. This ensures that complex calculations for filtering, grouping, and finding overlaps are only performed when the underlying data or user inputs (like search terms) change, optimizing performance. Multiple Data Structures: It calculates and provides data for three distinct views: overallGroupedData: A project-wide summary of all coded segments, grouped by code. detailedDataByDocument: A hierarchical view, grouping segments first by document, then by code. filteredOverlapsData: A specialized view that identifies and groups all instances of overlapping codes. State Management: Manages the state for user interactions within the tables, including search terms for different views (searchTerm, overlapSearchTerm) and the current sorting configuration (sortConfig). sortCodeGroups Helper: A utility function to apply sorting logic (by name or frequency) to the grouped data based on the current sortConfig.	
Usage	This custom hook is called once within the parent component responsible for displaying the data tables. The parent provides the raw project data as props, and the hook returns an object containing the processed data and state setters, which are then used to render the UI and connect controls like search inputs and sort buttons.	
Data Schema	This hook transforms raw project data into structured formats for display. ▪ Inputs (Props): codedSegments, codeDefinitions, project. ▪ Outputs (Returned Object): Includes state setters (setSearchTerm, setSortConfig) and derived data arrays like overallGroupedData,	

	detailedDataByDocument, and filteredOverlapsData, each with a specific nested structure for rendering.
.env Configuration	N/A
Error Handling	▪ This hook is focused on synchronous data transformation and does not contain explicit error handling. It is designed to be resilient to partially incomplete data by providing fallback values (e.g., for uncategorized segments).

2.4.10. layout

2.4.10.1. edit-mode

2.4.10.1.1. EditToolbar.jsx

Purpose	Provides a dedicated toolbar with tools for editing a document's text content, including save, undo/redo, text formatting, and find/replace functionalities.	
Dependencies	**Internal**	N/A
	External	react react-icons framer-motion
Key Components	Controlled Component: The toolbar is a fully controlled component; its state and behavior (e.g., input values, button disabled states, actions) are managed by a parent hook or component through an extensive set of props. Find and Replace UI: Features animated, collapsible panels for "Find" and "Find & Replace" operations, with controls for navigation and execution of replacements. Text Formatting Tools: Includes interactive controls for dynamically adjusting the font size and line height of the text editor. Informational Tooltip: A dismissible, animated tooltip provides users with a helpful tip on how to best format their documents for compatibility with other application features. Unsaved Changes Indicator: A visual indicator animates next to the "EDIT MODE" title to clearly inform the user when their document has unsaved changes.	

Usage	This component is designed to be rendered by a parent view (e.g., ProjectView) only when that view is in an "edit mode". The parent is responsible for providing all necessary state and callback functions as props to control the toolbar's functionality.
Data Schema	This component does not directly interact with any APIs. Its data interface consists of the numerous props it receives to control its state and the callback functions it invokes to signal user actions to its parent.
.env Configuration	N/A
Error Handling	▪ The component disables UI controls when they are not applicable (e.g., disabling the "Undo" button when canUndo is false), preventing invalid user actions. ▪ It uses useEffect to handle clicks outside of its popups (the formatting tip and line height dropdown) to close them automatically.

2.4.10.1.2. DialogueCard.jsx

Purpose	Renders a single, editable block of a transcript, displaying its timestamp, speaker, and dialogue, and providing functionality for in-place editing of the speaker and text.	
Dependencies	**Internal**	N/A
	External	react react-icons
Key Components	In-place Speaker Editing: A UI that allows the user to click on a speaker's name to enter an editing mode, complete with save and cancel actions. An useEffect hook automatically focuses the input field when editing begins. Content Editable Dialogue: The main dialogue text is rendered in a with the contentEditable attribute, allowing for direct, rich-text-like editing in the browser. Changes are saved on blur. Click-Outside-to-Cancel: An effect hook detects clicks outside the speaker editor and automatically cancels the editing process, preventing the UI from getting stuck in an edit state. Timestamp Interaction: The entire card is clickable while holding the Ctrl or Cmd key, which triggers a callback intended for seeking an associated audio or video file to that specific timestamp.	

Usage	This is a child component designed to be mapped over an array of dialogue blocks within a parent transcript editor. The parent component must provide the block data for rendering and all necessary callback functions (onDialogueChange, onInitiateRename, onTimestampClick) to handle state updates.
Data Schema	The component's interface is defined by its props: ▪ block Prop: An object with the shape { timestamp: string, speaker: string, dialogue: string }. ▪ onDialogueChange Callback: A function invoked with (index, newDialogueText). ▪ onInitiateRename Callback: A function invoked with (oldSpeakerName, newSpeakerName, index).
.env Configuration	N/A
Error Handling	▪ The component prevents saving an empty or whitespace-only speaker name. ▪ The click-outside handler for the speaker editor ensures a robust user experience by preventing the component from remaining in an edit state unintentionally.

2.4.10.1.3. TranscriptEditor.jsx

Purpose	Renders an interactive transcript editor that parses raw text into structured, editable dialogue blocks, and manages state for in-place editing of dialogue and speaker names.	
Dependencies	**Internal**	./DialogueCard.jsx ../../components/ConfirmationModal.jsx
	External	react
Key Components	Transcript Parsing: A useEffect hook contains a parseContent function that uses a regular expression to transform the raw content string prop into an array of structured DialogueBlock objects. Content Reconstruction: A reconstructContent function reverses the parsing process, converting the array of DialogueBlock objects back into a single string and notifying the parent component of the change via the onContentChange callback.	

	Speaker Rename Workflow: The component manages the entire workflow for renaming speakers. It identifies all instances of a speaker's name and uses a confirmation modal to ask the user whether to rename a single instance or all of them. Timestamp Interaction: Includes a parseTimestamp helper to convert timestamp strings (e.g., [00:01:23]) into seconds, which is then passed to the parent through the onTimestampClick prop for audio/video synchronization.
Usage	This is a controlled component designed to be used inside a larger view. A parent component must provide the raw content string and an onContentChange callback to manage the transcript's state. It renders a list of DialogueCard child components and orchestrates their state updates.
Data Schema	The component's interface is defined by its props: ▪ content Prop: The raw transcript text as a single string. ▪ onContentChange Callback: A function that receives the updated raw content string. ▪ onTimestampClick Callback: A function that receives a timeInSeconds (Number) value. ▪ Internal DialogueBlock Type: The component processes text into an internal array of objects with the shape { id, timestamp, speaker, dialogue }.
.env Configuration	N/A
Error Handling	▪ The parsing logic is designed to gracefully handle lines that do not match the standard transcript format by treating them as notes or continuations of previous dialogue. ▪ The speaker renaming process is handled through a confirmation modal to prevent accidental bulk changes.

2.4.10.2. AudioPlayer.jsx

Purpose	Provides a feature-rich, collapsible audio player with controls for playback, a draggable progress bar, volume, looping, and variable playback speeds, which can also be controlled externally.	
Dependencies	**Internal**	N/A
	External	react react-icons

Key Components	HTML5 Audio Integration: Uses a ref to directly control a native <audio> element, attaching event listeners (timeupdate, loadedmetadata, etc.) to synchronize the component's state with the audio's real-time status. Draggable Progress Bar: Implements a custom interactive scrubber that allows users to seek to any point in the audio by clicking and dragging on the progress bar. useImperativeHandle: Exposes a seekToTime(time) method to parent components via a ref. This is a key feature that allows an external component, like a transcript viewer, to programmatically control the player's position. Collapsible UI: The entire player can be hidden to save space and shown again with a click, with its state managed internally. OS-Aware Tooltips: Includes logic to detect the user's operating system to display the correct keyboard shortcut (Cmd for Mac, Ctrl for others) in informational tooltips.
Usage	This component is used by placing it within a parent view and providing the src URL of the audio file and a unique fileId. The fileId prop is crucial, as changing it will cause the player to fully reset for the new audio source. A parent can pass a ref to gain access to the seekToTime method.
Data Schema	The component's interface is defined by its props and exposed methods: ▪ src Prop (String, required): The URL of the audio source. ▪ fileId Prop (String, required): A unique identifier for the audio file. ▪ seekToTime(time) Method: An exposed function (via ref) that accepts a time in seconds (Number) to jump to.
.env Configuration	N/A
Error Handling	▪ A try...catch block around the audio.play() method handles potential browser errors related to autoplay policies. ▪ The component disables interactive controls until the audio metadata has loaded (isLoaded state), preventing user actions that could cause errors. ▪ useEffect cleanup functions are used to properly remove global event listeners, preventing memory leaks.

2.4.10.3. DocumentToolbar.jsx

Purpose	Provides a comprehensive toolbar for the document viewer, offering controls for text search, formatting, undo/redo, and a set of mutually exclusive tools for creating annotations.	
Dependencies	**Internal**	N/A
	External	react react-icons framer-motion
Key Components	Controlled Component: The toolbar is a fully controlled component. Its state (e.g., active tool, search query) and all its actions are managed by a parent component through an extensive set of props. Mutually Exclusive Annotation Tools: Provides a set of tool buttons (Code, Memo, Highlight, Erase) that are mutually exclusive. Only one tool can be active at a time, which is managed by the activeTool state prop. Integrated Search Bar: Includes a search input with controls to navigate between matches and clear the search, with all logic handled by the parent. Formatting Controls: Offers UI elements for adjusting the document's font size and line spacing. The line height control includes a custom dropdown with preset options and a field for user-defined values. Tool-Specific Dropdowns: The "Code" and "Highlight" tools have associated animated dropdowns for further configuration, such as toggling code visibility or selecting a highlight color.	
Usage	This component should be rendered by a parent view (like DocumentViewer or ProjectView). The parent is responsible for providing all the necessary state values and callback functions as props to manage the toolbar's behavior and respond to user interactions.	
Data Schema	This component does not interact with any APIs. Its data interface is defined by its extensive props, which include numerous state values (activeTool, viewerSearchQuery, fontSize) and handler functions (setActiveTool, handleViewerSearchChange, onUndo).	
.env Configuration	N/A	

Error Handling	▪ The toolbar disables buttons when their corresponding actions are unavailable (e.g., undo/redo buttons, search navigation), providing clear visual feedback to the user. ▪ It uses a useEffect hook to handle clicks outside of its dropdowns, ensuring they close automatically for a clean user experience.

2.4.10.4. DocumentViewer.jsx

Purpose	Serves as the primary interface for displaying and interacting with document content, featuring a sophisticated rendering engine that overlays annotations (codes, highlights, memos) and provides distinct modes for viewing and editing.	
Dependencies	**Internal**	./CodeTooltip.jsx ./DocumentToolbar.jsx ./edit-mode/TranscriptEditor.jsx
	External	react react-icons framer-motion
Key Components	Annotation Rendering Engine: The core of the component is a renderAnnotatedFragment function. It takes raw text and an array of annotations, calculates all overlapping boundaries, and dynamically wraps text fragments in styled elements. This allows for multiple layers of annotations (e.g., a code, a highlight, and a search result) to be visually applied to the same text. Conditional View Modes: The component intelligently switches between different renderers based on props: ▪ Rich View Mode: Displays text with all visual annotations and interactive markers. ▪ Transcript Edit Mode: Renders the specialized TranscriptEditor component if the document has audio. ▪ Plain Text Edit Mode: Renders a standard <textarea> for editing regular text documents. Interactive Markers: When rendering annotations, it adds small, interactive icons (e.g., <sup> tags for codes, <FaStickyNote> for memos) that allow users to click to activate, hover to see toolbars, or right-click for more options.	

	DocumentToolbar Integration: Renders and controls the main toolbar, passing down a large number of state props and callbacks to manage its functionality.
Usage	This is a large, controlled component that acts as the central panel of the ProjectView. It should be given a ref and is almost entirely managed by a parent hook (useProjectViewHooks), which provides its content, annotation data, and all necessary event handlers as props.
Data Schema	This component's data interface is its props. Key data props include: ▪ selectedContent: The raw text string of the document to display. ▪ codedSegments, inlineHighlights, memos: Arrays of annotation objects, each with a required structure including startIndex and endIndex. ▪ viewerSearchMatches: An array of search match objects.
.env Configuration	N/A
Error Handling	▪ The component includes a fallback UI to prompt the user to select a document if no content is provided. ▪ Its rendering logic is designed to be resilient, with default values for potentially missing data within annotation objects (e.g., a fallback color for a code). ▪ All interactive logic that could fail (e.g., saving an annotation) is handled by callback props passed down from the parent.

2.4.10.5. FloatingToolbar.jsx

Purpose	Renders a compact, floating toolbar at specific screen coordinates to provide users with immediate, contextual actions for a text selection, such as coding, creating a memo, or highlighting.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	Absolute Positioning: The component uses x and y props to apply fixed positioning, allowing it to appear anywhere on the screen, typically next to the user's text selection.	

	Callback-Driven Actions: It is a purely presentational component where each button's onClick event is tied to a callback function (onCode, onMemo, onHighlight, onCancel) provided by the parent. framer-motion Animations: Utilizes <AnimatePresence> and <motion.div> to create a smooth fade-and-slide animation when the toolbar appears and disappears.
Usage	This is a controlled component intended to be rendered by a parent that manages text selection events. The parent component is responsible for calculating the x and y coordinates for positioning the toolbar and must provide the callback functions to handle user actions.
Data Schema	N/A
.env Configuration	N/A
Error Handling	▪ The component uses e.stopPropagation() to prevent clicks on the toolbar from unintentionally affecting the underlying document content (e.g., deselecting text). ▪ It relies entirely on the parent component to provide valid callback functions for its actions.

2.4.10.6. ImportOptionsModal.jsx

Purpose	Provides a guided, multi-step modal to help users import files, allowing them to first select the file type and then choose a content processing option before uploading.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	Multi-Step Workflow: The modal operates as a wizard, using a modalStep state to guide the user through a sequence of choices: initial type selection (audio/text), then content format options (e.g., split by turn or by sentence). Hidden File Inputs: It uses hidden <input type="file"> elements that are programmatically triggered. This allows for a fully custom and user-friendly UI for the file selection process.	

	Client-side Validation: Includes a check to validate the size of the selected file against a predefined maximum before passing it to the handler, providing immediate feedback to the user if a file is too large. Callback-Driven: The component is controlled by a parent and uses callback props (handleAudioImport, handleTextImport) to pass the selected file and processing options back for handling.
Usage	This is a controlled component. A parent must manage the show prop to control its visibility. The parent also provides the onClose handler and the handleAudioImport and handleTextImport functions, which will receive the file and the chosen splitting option from the modal.
Data Schema	This component does not interact with APIs. It communicates with its parent via callbacks: ▪ handleAudioImport(file, splitOption): Called with the audio File object and the chosen format string. ▪ handleTextImport(file, splitOption): Called with the text File object and the chosen format string.
.env Configuration	N/A
Error Handling	▪ The component performs client-side file size validation and uses a browser alert() to notify the user if a file exceeds the maximum allowed size. ▪ It relies on the parent component to handle any errors that may occur during the actual backend file processing.

2.4.10.7. LeftPanel.jsx

Purpose	Provides a collapsible and resizable side panel that acts as the primary hub for managing a project's assets, featuring a global search and distinct sections for files, code definitions, coded segments, and memos.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons

Key Components	Collapsible and Resizable UI: The panel's width can be dynamically adjusted by the user dragging its edge, and it can be fully collapsed or expanded with a toggle button. Global Search: A single search input at the top filters the contents of all sections (files, codes, segments, and memos) simultaneously. Collapsible Sections: The UI is organized into four main sections, each of which can be independently expanded or collapsed to manage screen real estate. In-place Renaming: The files list allows for renaming a file directly in the list without opening a separate modal. Context Menus: Each item in the lists (files, codes, etc.) features context-aware controls that appear on hover or click, providing actions like edit, delete, and pin. Controlled Component: The panel is a fully controlled component; its content, state, and all user interactions are managed by a parent hook via an extensive set of props.
Usage	This component is designed to be the main side panel within the ProjectView. It is not self-contained and must be controlled by a parent that provides its data and a large number of callback functions for every user action (e.g., selecting a file, deleting a code).
Data Schema	This component's data interface is defined by its props. Key data props include: ▪ project: The full project object. ▪ codeDefinitions: An array of code definition objects. ▪ groupedCodedSegments: A pre-grouped array of coded segments. ▪ groupedMemos: A pre-processed array of memos. ▪ pinnedFiles: An array of file IDs.
.env Configuration	N/A
Error Handling	▪ The component provides clear feedback for empty states, such as "No files found" or "No codes defined yet." ▪ It disables all interactions with an overlay when the parent view is in isEditing mode to prevent conflicting actions. ▪ It relies on the parent component to handle and display any errors that result from the callback functions it invokes (e.g., an error during a file deletion).

2.4.10.8. Navbar.jsx

Purpose	Renders the application's main navigation bar, providing contextual controls based on the current view, a theme toggler, and a user profile dropdown for authenticated users.	
Dependencies	**Internal**	../theme/ThemeToggle.jsx ../auth/AuthContext.jsx ../theme/Logo.jsx
	External	react react-router-dom framer-motion react-icons
Key Components	Contextual Controls: The navbar uses the useLocation hook to detect if the user is on a project-specific page (isProjectView). It conditionally renders additional project-related action buttons (e.g., "New Project", "Project Overview") only when relevant. User Profile Dropdown: An animated dropdown menu for authenticated users that displays user information and provides links for preferences, bug reports, feedback, and logging out. Navigation Interception: The handleNavigation and handleLogout functions are designed to be interruptible. They check for optional props (onNavigateAttempt, onLogoutAttempt) which, if provided, allow a parent component to intercept the action and show an "unsaved changes" confirmation before proceeding. Click-Outside-to-Close: A useEffect hook manages the profile dropdown's visibility, automatically closing it when the user clicks anywhere outside of the dropdown menu.	
Usage	This component is a top-level layout element intended for pages that require global navigation. It is controlled by a parent component that provides various callback functions to handle actions like opening modals and managing navigation interception logic.	
Data Schema	This component does not directly interact with APIs. Its main data source is the user object from the AuthContext, which it uses to display the user's name and email in the profile dropdown.	

.env Configuration	VITE_GOOGLE_FORM_URL: The URL for the "Report a Bug" link. VITE_FEEDBACK_FORM_URL: The URL for the "Give Feedback" link.
Error Handling	▪ The component disables certain buttons (e.g., "Project Overview") when the isEditing prop is true to prevent users from accidentally losing unsaved work. ▪ The handleLogout function includes a fallback mechanism to ensure logout functionality even if the more complex confirmation modal props are not provided.

2.4.10.9. PreferencesModal.jsx

Purpose	Renders a modal for managing user-specific application preferences, including UI settings like tooltip visibility and actions to restore default settings.	
Dependencies	**Internal**	N/A
	External	react react-icons framer-motion
Key Components	Controlled Component: The modal's visibility and all preference states are managed by a parent component through props. Preference Toggles: Includes UI elements like a styled toggle switch to allow users to enable or disable specific features (e.g., "Show Code Tooltip on Hover"). Action Buttons: Provides buttons for global actions like "Restore to Defaults," which trigger callbacks passed from the parent. framer-motion Integration: Uses <AnimatePresence> and <motion.div> to provide smooth fade and scale animations when the modal is opened or closed.	
Usage	This is a controlled component. A parent component must manage the show state to control its visibility and provide the current state of each preference (e.g., showTooltip) as well as the callback functions to handle user actions (onClose, onToggleTooltip, onRestoreDefaults).	
Data Schema	This component does not interact with any APIs. Its data interface is defined entirely by its props: ▪ show (Boolean): Controls visibility. ▪ showTooltip (Boolean): The current value for the tooltip preference.	

	▪ onClose, onToggleTooltip, onRestoreDefaults: Callback functions to notify the parent of user actions.
.env Configuration	N/A
Error Handling	▪ The component relies on the parent to provide valid callback functions. ▪ It uses e.stopPropagation() on the main modal content to prevent clicks inside the modal from accidentally triggering the onClose handler on the backdrop.

2.4.10.10. ProjectView.jsx

Purpose	Serves as the main workspace for a single project, assembling and orchestrating all major UI components like the navigation bar, side panel, document viewer, and all modals.	
Dependencies	**Internal**	../layout/Navbar.jsx ../components/ConfirmationModal.jsx ../code/DefineCodeModal.jsx ../memo/MemoModal.jsx ../layout/FloatingToolbar.jsx ../code/FloatingAssignCode.jsx ../memo/FloatingMemoInput.jsx ../table/CodedSegmentsTableModal.jsx ../layout/LeftPanel.jsx ../layout/DocumentViewer.jsx ./ImportOptionsModal.jsx ../hooks/Hooks.jsx ../layout/AudioPlayer.jsx ../layout/edit-mode/EditToolbar.jsx ../code/CodeDetailsModal.jsx ./PreferencesModal.jsx

		../code/SplitMergeCodesModal.jsx ../code/SplitReviewModal.jsx ../auth/AuthContext.jsx
	External	react react-router-dom axios file-saver
Key Components	useProjectViewHooks Integration: The component's architecture is centered around the useProjectViewHooks custom hook. It calls this hook to receive a single, comprehensive object containing nearly all the state and handler functions required for the entire view. Component Composition: Acts as the master layout component that renders all other parts of the project interface, such as Navbar, LeftPanel, and DocumentViewer. It is also responsible for rendering all modals and controlling their visibility based on state from the hook. Prop Drilling: Its primary role is to destructure the large object from useProjectViewHooks and pass the relevant state and functions down as props to the appropriate child components. Edit Mode Management: Contains the local state and logic for the document text editor (fileInEditMode, editedContent), including a local undo/redo history that is separate from the main annotation history. Data Loss Prevention: Implements several useEffect hooks that add event listeners to the window (beforeunload, wheel) to prevent accidental navigation or browser closing when there are unsaved changes in edit mode.	
Usage	This is a page-level component that is rendered by the main application router (App.jsx) when a user navigates to a URL matching the /project/:id pattern. It represents the entire interactive workspace for a project.	
Data Schema	This component does not directly interact with any APIs. All API communication and data schema handling are abstracted away into the useProjectViewHooks hook.	
.env Configuration	This component does not directly use environment variables, but it renders child components (like AudioPlayer) and uses a hook (useProjectViewHooks) that depend on VITE_BACKEND_URL.	

Error Handling	▪ Loading and Error States: It handles the primary loading and error states for the page. It displays a "Loading..." message while the initial project data is being fetched and will render a full-page error message if the fetch fails. ▪ Navigation Guards: It provides handleNavigationAttempt and handleLogoutAttempt functions to the Navbar to intercept navigation and prompt the user for confirmation if they have unsaved changes in edit mode.

2.4.11. memo

2.4.11.1. FloatingMemoInput.jsx

Purpose	Renders a floating form at a specified screen position that allows users to create a new memo with a title and content, including client-side validation.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	Absolute Positioning: The component uses x and y props to apply fixed positioning, allowing it to appear anywhere on the screen, typically near a user's text selection. Client-side Validation: The handleSubmit function validates the form before saving, checking for required fields (title, content) and ensuring the new memo title is not a duplicate of an existing one. State Management: Manages the state for the title and content inputs, as well as a local error state to display validation messages directly to the user. Callback-Driven: It is a controlled component that communicates with its parent via onSave and onClose callbacks to handle its logic and dismissal.	
Usage	This is a controlled component rendered by a parent that manages text selections. The parent component is responsible for controlling its visibility, calculating the x and y coordinates, and providing the onSave callback to handle the creation of the new memo.	
Data Schema	This component's interface is defined by its props. ▪ allMemos Prop: An array of existing memo objects, used for duplicate title validation.	

	▪ onSave Callback: An async function that is called with the new memo's data: { title, content }.
.env Configuration	N/A
Error Handling	▪ Validation Feedback: Displays specific error messages to the user for validation failures, such as a missing title or a duplicate title. ▪ Asynchronous Errors: It wraps the onSave callback in a try...catch block, allowing it to catch and display errors that may occur during the parent's save operation (e.g., a failed API call). ▪ Event Propagation: Uses e.stopPropagation() to prevent clicks within the form from unintentionally interacting with the document underneath.

2.4.11.2. MemoModal.jsx

Purpose	Provides a multi-functional modal for creating, viewing, editing, and deleting memos, adapting its UI and behavior based on whether it is creating a new memo or interacting with an existing one.	
Dependencies	**Internal**	../auth/AuthContext.jsx
	External	react framer-motion react-icons
Key Components	Multi-modal Behavior: The component operates in different modes. It starts in "create" mode if no initialMemo is provided, or in "view" mode if one is. The user can switch from "view" to "edit" mode. State Initialization: A useEffect hook populates the modal's form fields and sets its initial mode (view or create) whenever it is shown, based on the initialMemo and selectionInfo props. Client-side Validation: Before saving, the handleSubmit function validates that the title and content are not empty and checks against the allMemos prop to prevent duplicate titles. Authorship: It integrates with useAuth to automatically include the current user's name and ID when saving a memo, ensuring proper attribution.	
Usage	This is a controlled component. A parent must manage the show state. To create a new memo, provide the selectionInfo, onSave, and onClose props. To	

	view or edit an existing memo, provide the initialMemo object along with the onSave, onClose, and onDelete callbacks.
Data Schema	The component communicates with its parent via callbacks: ▪ onSave Callback: Receives a memo data object containing title, content, text, author, authorId, and an _id if editing. ▪ onDelete Callback: Receives the memoId and memoTitle of the memo to be deleted. ▪ initialMemo Prop: Expects a memo object with properties like _id, title, content, text, etc.
.env Configuration	N/A
Error Handling	▪ Validation Feedback: Displays clear error messages for required fields or duplicate titles. ▪ Asynchronous Errors: Wraps the onSave callback in a try...catch block to catch and display any errors that occur during the parent's save operation (e.g., a failed API call). ▪ Dismissal: The modal can be closed by clicking the backdrop or the close button, which triggers the onClose callback.

2.4.12. Project

2.4.12.1. CreateProjectModal.jsx

Purpose	Renders a modal with a form that allows users to create a new project by providing a name and an optional description.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	State Management: Uses the useState hook to manage the form's input fields (name, description) and an error state for displaying validation messages. handleSubmit(e): The form submission handler that performs client-side validation to ensure a project name is provided before calling the parent's onConfirm callback with the new project data.	

	handleClose(): A cleanup function that resets the modal's internal state (clearing inputs and errors) before calling the parent's onClose function. framer-motion Animations: Uses <AnimatePresence> and <motion.div> to provide smooth fade and scale animations when the modal appears and disappears.
Usage	This is a controlled component. A parent component must manage the show state to control its visibility and provide onClose and onConfirm callback functions to handle its dismissal and form submission.
Data Schema	This component does not interact directly with APIs. It passes a data object to the parent component via the onConfirm callback with the following structure: onConfirm Payload: ▪ name (String, required) ▪ description (String, optional)
.env Configuration	N/A
Error Handling	▪ Client-side Validation: The component checks if the project name is empty and displays an inline error message if it is, preventing form submission. ▪ Dismissal: The modal can be safely closed by clicking the backdrop or the close button, which triggers the handleClose function to reset its state. ▪ Event Propagation: Uses e.stopPropagation() on the modal content to prevent clicks inside the modal from accidentally triggering the onClose handler on the backdrop.

2.4.12.2. EditProjectModal.jsx

Purpose	Renders a modal containing a form to edit the name of an existing project.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	State Management: Uses the useState hook to manage the project name input field and an error state for validation messages.	

	useEffect Hook: Populates the form with the current project's name when the modal is opened or the project data changes. handleSubmit(e): The form submission handler that performs client-side validation to ensure the new name is not empty before passing the updated data to the parent via the onConfirm callback. framer-motion Animations: Uses <AnimatePresence> and <motion.div> to provide smooth fade and scale animations when the modal appears and disappears.
Usage	This is a controlled component. A parent component must manage the show prop to control its visibility, provide the project object to be edited, and supply the onClose and onConfirm callback functions.
Data Schema	This component does not interact directly with APIs. It interfaces with its parent via props and callbacks: ▪ project Prop: Expects a project object containing at least a name property. ▪ onConfirm Callback: Is called with a data object containing the updated fields: { name: 'New Project Name' }.
.env Configuration	N/A
Error Handling	▪ Client-side Validation: The component checks if the project name is empty upon submission and displays an inline error message if it is. ▪ Dismissal: The modal can be safely closed by clicking the backdrop or the close button, which triggers the onClose callback. ▪ Event Propagation: Uses e.stopPropagation() on the modal content to prevent clicks inside the modal from accidentally triggering the onClose handler on the backdrop.

2.4.12.3. Projects.jsx

Purpose	Renders the main projects dashboard, allowing users to view, search, sort, create, edit, copy, and delete their projects.	
Dependencies	**Internal**	../layout/Navbar.jsx ../auth/AuthContext.jsx ../components/ConfirmationModal.jsx ./EditProjectModal.jsx

		./CreateProjectModal.jsx
	External	react react-router-dom framer-motion axios react-icons
Key Components	Data Fetching and State Management: Fetches all user projects on mount and manages a comprehensive local state, including the list of projects, UI states (view mode, sort order), search queries, and the visibility of all modals. Client-side Filtering and Sorting: A useMemo hook performantly filters and sorts the project list based on the user's search query and selected sort configuration, providing a fast and responsive UI without re-fetching data from the server. CRUD Handlers: Contains all async functions (handleCreateProject, handleUpdateProject, handleDeleteProject, handleCopyProject) for making API calls to the backend to manage projects. These handlers update the local state upon success to ensure the UI reflects changes immediately. Modal Orchestration: Renders and controls all necessary modals for project management, including separate components for creating and editing, and a configurable ConfirmationModal for handling destructive actions like deleting and copying.	
Usage	This is a page-level component that is protected by PrivateRoute and is rendered by the main application router when a user navigates to the /projects path. It serves as the central dashboard for all of a user's projects. GET /api/projects/my-projects: Fetches an array of project objects. POST /api/projects/create: Sends a { name, description } object. PUT /api/projects/:id: Sends an { name } object. DELETE /api/projects/:id: Sends a request with no body. POST /api/projects/:id/copy: Sends a { includeAnnotations } object.	
Data Schema	This component directly handles all API requests for creating, reading, updating, and deleting projects.	

	_id: string - The unique identifier for the project. name: string - The name of the project. description: string - An optional description for the project. createdAt: Date - The date the project was originally created. updatedAt: Date - The date the project was last modified. includeAnnotations: Boolean- Flag used while creating a duplicate of an existing project.
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoints for all API requests.
Error Handling	▪ API Errors: All API interactions are wrapped in try...catch blocks. If a request fails, the error message from the backend is captured and displayed to the user in an alert banner. ▪ Confirmation for Destructive Actions: It uses a ConfirmationModal for delete and copy operations to prevent accidental data loss. The delete action requires the user to type the project's name as an additional safeguard. ▪ Empty/Loading States: The component displays a loading spinner while fetching initial data and a user-friendly message if no projects are found.

2.4.13. stats

2.4.13.1. chi-squared

2.4.13.1.1. ChiSquareControlPanel.jsx

Purpose	Provides a set of UI panels for configuring the parameters required for different types of Chi-Square statistical tests: Independence, Homogeneity, and Goodness-of-Fit.	
Dependencies	**Internal**	../../components/SearchableMultiSelectDropdown.jsx ../../components/SearchableMultiCodeDropdown.jsx
	External	react react-icons

Key Components	ChiSquareControlPanel: The main exported component that acts as a router. It conditionally renders one of the three specific configuration panels based on the testType prop it receives. IndependencePanel: A form with two multi-select dropdowns for selecting the codes and documents that will form the rows and columns of a contingency table. HomogeneityPanel: A form for defining and populating multiple, distinct groups of documents to compare code distributions across them. It intelligently disables documents in other dropdowns once they've been selected for a group. GoodnessOfFitPanel: A form that allows users to select codes and then specify an expected frequency distribution, either "uniform" or "custom". The custom option includes inputs for defining specific percentages and validates that they sum to 100%.
Usage	The ChiSquareControlPanel is a controlled component intended for use within a larger statistics view. A parent component must provide the testType prop to determine which panel to display, as well as all the necessary state values and setter functions to manage the inputs within that panel.
Data Schema	This component is purely for gathering user input. Its data interface is defined by its props: ▪ testType Prop: A string that determines which panel is rendered. ▪ Data Props: Receives arrays like codeDefinitions and project.importedFiles to populate dropdowns. ▪ State & Callback Props: Receives numerous state values and their setters from a parent hook to function as a controlled component (e.g., indepCodes, setIndepCodes).
.env Configuration	N/A
Error Handling	▪ The component includes real-time, client-side validation to guide the user. For instance, the GoodnessOfFitPanel displays the running total of custom proportions and highlights it if it does not equal 100%. ▪ It relies on the parent component/hook to handle more complex validation, such as ensuring the minimum number of items has been selected before a test can be run.

2.4.13.1.2. ChiSquareDisplay.jsx

Purpose	Provides a set of data visualization components that render appropriate bar charts for different types of Chi-Square test results, making the statistical output easier to interpret.	
Dependencies	**Internal**	N/A
	External	react recharts
Key Components	ChiSquareDisplay: The main exported component that acts as a router. It inspects the subtype property of the results object and conditionally renders the correct chart for the given test. GoodnessOfFitChart: A component that renders a grouped bar chart to visually compare the observed frequencies against the expected frequencies for a Goodness-of-Fit test. ContingencyChart: A component that visualizes contingency table data. Its key feature is a toggle switch that allows the user to dynamically switch the view between a grouped and a stacked bar chart. truncateLabel: A utility function that shortens long x-axis labels to prevent them from overlapping and ensure the chart remains readable.	
Usage	This is a presentational component. A developer should render the main ChiSquareDisplay component and pass the results object, which is received from the statistical analysis API, as a prop. The component will then automatically handle the rendering of the correct chart.	
Data Schema	The components' data interface is the results prop, which must have a specific structure depending on the test subtype: ▪ For Goodness-of-Fit: The results object must contain observedCounts, expectedCounts, and categoryLabels. ▪ For Independence or Homogeneity: The results object must contain observedTable (a 2D array), rowLabels, and colLabels.	
.env Configuration	N/A	
Error Handling	▪ The main ChiSquareDisplay component returns null if the results prop is missing or if the subtype is not recognized, preventing rendering errors.	

	▪ The ContingencyChart gracefully handles a dynamic number of data series by mapping over the provided labels.

2.4.13.1.3. ChiSquareDistributionChart.jsx

Purpose	Renders a probability density function curve for a Chi-Square distribution to visually represent the results of a statistical test, highlighting the test statistic and the p-value area.	
Dependencies	**Internal**	N/A
	External	react recharts jstat
Key Components	Dynamic Data Generation: A useMemo hook uses the jStat library to dynamically calculate the data points for the Chi-Square probability density function curve based on the provided degrees of freedom. recharts Integration: It uses a <ComposedChart> from the recharts library to combine multiple chart types: ▪ A <Line> component to draw the distribution curve. ▪ An <Area> component to shade the critical region corresponding to the p-value. ▪ A <ReferenceLine> to mark the position of the calculated test statistic on the x-axis. CustomTooltip: A small helper component to provide custom formatting for the data that appears when a user hovers over the chart.	
Usage	This is a presentational component used to visualize statistical results. A parent component should render it and pass the required df, statistic, and pValue from a completed Chi-Square test as props.	
Data Schema	The component's interface is defined by its props: ▪ df (Number, required): The degrees of freedom for the distribution. ▪ statistic (Number, required): The calculated Chi-Square value from the test. ▪ pValue (Number, required): The p-value from the test.	
.env Configuration	N/A	

Error Handling	The component returns null and does not render if the essential props (df and statistic) are not provided, preventing potential crashes.

2.4.13.1.4. ChiSquareTypeSelector.jsx

Purpose	Renders a selection screen that educates the user about different Chi-Square test types, using contextual examples from their own data to help them choose the appropriate test.	
Dependencies	**Internal**	N/A
	External	react react-icons
Key Components	ChiSquareTypeSelector: The main component that orchestrates the layout and provides data to the child cards. ChiSquareTypeCard: A reusable, presentational component that displays the details for a single test type, including an icon, title, description, and an example. Contextual Examples: The component takes a stats prop with project summary data (e.g., total number of coded segments, most frequent codes) and injects this data into the example text on each card, making the choice more intuitive for the user.	
Usage	This is a presentational component. A parent component renders it and provides a stats object with project data and an onSelect callback function. When a user clicks a card, the onSelect callback is fired with the key for the chosen test type (e.g., 'goodness-of-fit'), allowing the parent to advance to the next step in the UI flow.	
Data Schema	The component's data interface is defined by its props: ▪ stats Prop: An object containing project summary data like numSegments and mostUsedCode. ▪ onSelect Callback: A function that is invoked with a single string argument representing the selected test type.	
.env Configuration	N/A	
Error Handling	▪ The component gracefully handles missing or incomplete stats data by providing fallback placeholder text in the examples.	

	▪ The ChiSquareTypeCard can be disabled via a prop, which makes it unclickable and visually distinct.

2.4.13.1.5. ObservedFrequencyTable.jsx

Purpose	Provides components for displaying observed frequency data from statistical tests in a clear, tabular format, with specific layouts for different Chi-Square tests.	
Dependencies	**Internal**	N/A
	External	react
Key Components	ObservedFrequencyTable: The main exported component that acts as a router. It inspects the subtype property of the results object and conditionally renders the correct table for the given test. GoodnessOfFitTable: Renders a simple two-column frequency table, displaying each category and its observed frequency, along with a total count. ContingencyTable: Renders a 2D contingency table for Independence or Homogeneity tests. It dynamically creates rows and columns and automatically calculates and displays row totals, column totals, and the grand total.	
Usage	This is a presentational component. A developer should render the main ObservedFrequencyTable component and pass the results object from a completed statistical test as a prop. The component will then automatically display the correct table format.	
Data Schema	The component's data interface is the results prop, which must have a specific structure depending on the test subtype: ▪ For Goodness-of-Fit: The results object must contain observedCounts and categoryLabels. ▪ For Independence or Homogeneity: The results object must contain observedTable (a 2D array), rowLabels, and colLabels.	
.env Configuration	N/A	
Error Handling	▪ The main ObservedFrequencyTable component returns null if the results prop is missing or if the test subtype is not recognized, preventing rendering errors. ▪ The table components rely on the parent to provide props with the correct data structure.	

2.4.13.2. CombineCategoriesModal.jsx

Purpose	Provides a modal for combining multiple categories (e.g., codes) into larger groups, primarily to help users resolve statistical assumption violations like low expected frequencies.	
Dependencies	**Internal**	N/A
	External	react framer-motion react-icons
Key Components	Dynamic Group Management: Allows users to create and delete groups of codes. When a group is created, its constituent codes are removed from the list of available codes to prevent them from being used in multiple groups. State Management: Manages the state of the newly created groups as well as the list of availableCodes that have not yet been assigned to a group. Real-time Validation: A useEffect hook provides instant feedback to the user, checking if a proposed group name is already in use and displaying an error message. State Reset: A useEffect hook resets the modal's entire internal state whenever it is opened, ensuring a clean workspace for the user each time.	
Usage	This is a controlled component, typically shown after a statistical validation check fails. The parent component must control its visibility with the show prop and provide the original category details from the validation step. The parent also provides an onApply callback, which receives the array of new groups to trigger a re-validation.	
Data Schema	This component is prop-driven and communicates with its parent via a callback. ▪ details Prop: An object from a validation step, containing originalRowLabels and originalCodeIds. ▪ onApply Callback: This function is invoked with an array of group objects. Each object has the shape { newName: string, originalCodeIds: string[], originalNames: string[] }.	
.env Configuration	N/A	

Error Handling	▪ Client-side Validation: Prevents the creation of invalid groups (e.g., a group with fewer than two codes or a duplicate name) and disables the final "Apply" button until at least one valid group has been created. ▪ UI Feedback: Displays specific error messages to the user for invalid inputs. ▪ The component's state logic ensures a code cannot be assigned to more than one group at a time.

2.4.13.3. ExpectedFrequencyDetails.jsx

Purpose	Renders a detailed, educational breakdown of the calculations behind a Chi-Square test, displaying tables with both observed and expected frequencies alongside the formulas used.	
Dependencies	**Internal**	N/A
	External	react
Key Components	ExpectedFrequencyDetails: The main exported component that acts as a router, inspecting the subtype prop to render the appropriate detailed view for the given test. ContingencyTableDetails: Displays a full contingency table that shows not only the observed and expected values in each cell but also the explicit formula used to calculate the expected value (e.g., (Row Total × Col Total) / Grand Total). GoodnessOfFitDetails: Renders a frequency table that shows the observed and expected counts for each category. It also displays the formula used for the expected count, adapting it for uniform vs. custom distributions. Conditional Formatting: Both table components highlight cells in red where the calculated expected frequency is less than 5, immediately drawing the user's attention to potential violations of statistical assumptions.	
Usage	This is a presentational component intended for use within a statistical validation view. A developer renders the main ExpectedFrequencyDetails component and passes the details object from the validation results and the test subtype as props.	
Data Schema	The component's interface is defined by its props, which expect a specific structure: ▪ subtype Prop: A string indicating the test type (e.g., 'goodness-of-fit').	

	▪ details Prop: An object containing the calculated data from the validation step, such as observed, expected, rowTotals, etc
.env Configuration	N/A
Error Handling	▪ Each sub-component checks for the existence of the required details data and renders a fallback message if it's not available. ▪ The component's primary role is to help users understand a statistical "error" or warning (i.e., a violated assumption), so its core design is centered on clarifying these issues.

2.4.13.4. StatsModal.jsx

Purpose	Provides a comprehensive, multi-view modal for conducting statistical analyses, guiding the user through selecting a test, configuring variables, checking assumptions, and viewing results.	
Dependencies	**Internal**	./auth/AuthContext.jsx ./chi-squared/ChiSquareTypeSelector.jsx ./chi-squared/ChiSquareControlPanel.jsx ./StatsResultsPanel.jsx
	External	react framer-motion axios react-icons
Key Components	State Machine & View Router: The component operates as a state machine controlled by a view state. A renderContent function acts as a router, conditionally rendering different child components (ChiSquareTypeSelector, ChiSquareControlPanel, StatsResultsPanel) for each step of the workflow. Centralized State Management: Consolidates all state for the statistical analysis process, including user selections for each test type, loading and error states, API results, and the state for various confirmation modals. API Interaction Logic: Contains the async functions to communicate with the backend stats API. This includes functions for both validating statistical assumptions and for running the final test.	

	Assumption Confirmation Workflow: Implements a multi-step confirmation process that requires users to acknowledge key statistical assumptions (like independence of observations) before they can proceed with validation. Client-side Input Validation: A memoized value (areInputsIncomplete) provides real-time validation of user inputs, disabling action buttons until all necessary selections for a given test have been made.
Usage	This is a top-level modal component, likely rendered within the ProjectView. A developer controls its visibility with the show prop and provides the necessary project-wide data: the project object, codeDefinitions, and projectId.
Data Schema	This component is the primary client for the /api/stats/run backend endpoint. It constructs and sends JSON payloads whose structure is determined by the selected test type, including fields like codes, docList, indepDocs, and homoDocGroups.
.env Configuration	VITE_BACKEND_URL: The base URL for the backend API, used to construct the endpoint for all statistical analysis requests.
Error Handling	▪ API Errors: All API requests are wrapped in try...catch blocks. If a request fails, the error message from the backend is stored in an error state and displayed to the user in the results panel. ▪ User Workflow: It disables action buttons when inputs are incomplete to guide the user. It also uses confirmation modals to ensure users understand key statistical concepts before proceeding. ▪ State Reset: It uses useEffect to automatically clear previous validation and results data whenever a user changes their input selections, preventing stale or misleading information from being shown.

2.4.13.5. StatsResultsPanel.jsx

Purpose	Serves as the main display panel for the statistical analysis workflow, conditionally rendering the appropriate UI for each stage, including an assumption checklist, loading indicators, error messages, and a full report of the final test results with tables and charts.	
Dependencies	**Internal**	../theme/ThemeContext.jsx ./chi-squared/ChiSquareDisplay.jsx ./ExpectedFrequencyDetails.jsx ./chi-squared/ObservedFrequencyTable.jsx

		./chi-squared/ChiSquareDistributionChart.jsx ./CombineCategoriesModal.jsx
	External	react html-to-image recharts react-icons
Key Components	StatsResultsPanel: The main component that acts as a view router. It takes the current state of the analysis (results, isLoading, error, validationStatus) and renders the corresponding UI. ChartWithExport: A higher-order component that wraps a chart, adding an "Export as PNG" button with selectable quality options. It uses the html-to-image library to capture the chart. AssumptionChecklist: A component that displays a checklist of statistical assumptions and their validation status, with expandable sections to show detailed feedback and calculations. ActionButtons: A smart component that conditionally renders the correct action button based on the validation status, prompting the user to "Validate Data," "Run Test," or take corrective action like "Combine Categories." Results Display: When results are available, it renders a full report including key statistics, tables, and interactive charts.	
Usage	This is a presentational component intended to be the main content area of the StatsModal. It is fully controlled by a parent hook (like useStatsLogic), which provides all the necessary data and callback functions as props to drive its state and behavior.	
Data Schema	The component's interface is defined by its props: ▪ results Prop: A complex object containing the final data from a statistical test (e.g., statistic, pValue, interpretation). ▪ validationStatus Prop: An object where keys are assumption names (e.g., expectedFrequency) and values are objects containing the status and message for that assumption.	
.env Configuration	N/A	

Error Handling	▪ The component explicitly renders a dedicated UI for the isLoading and error states passed down from its parent. ▪ The ChartWithExport wrapper includes a try...catch block to handle errors during the image generation process, alerting the user if an export fails.

2.4.14. table

2.4.14.1. ChartRenderer.jsx

Purpose	Provides a single, versatile component that renders various chart types (Bar, Pie, Radar, Treemap, Word Cloud) based on a prop, centralizing visualization logic and enhancing charts with custom rendering components.	
Dependencies	**Internal**	./D3WordCloud
	External	react recharts framer-motion react-icons
Key Components	ChartRenderer: The main exported component that acts as a router, conditionally rendering the correct chart component based on the selectedChart prop. It also manages an animation loading state for the parent. Custom recharts Components: It includes several custom helper components to enhance the recharts library: ▪ CustomAxisTick: Implements word wrapping for long X-axis labels to improve readability. ▪ renderCustomizedLabel: Renders percentage values directly inside the slices of a Pie chart. ▪ CustomTreemapContent: A custom renderer for the Treemap chart that handles text wrapping and styling to ensure labels fit within their cells. D3WordCloud Integration: It renders the D3WordCloud component and provides a dedicated "Refresh" button that forces the word cloud to re-calculate its layout.	
Usage	This is a presentational component. A developer places it in a UI and controls it via props. The selectedChart prop determines which visualization is displayed, and the chartData prop provides the necessary data. It also requires	

	isDarkMode for styling and an setIsChartAnimating callback to communicate its animation state to the parent.
Data Schema	The component's interface is defined by its props: ▪ selectedChart Prop: A string that must be one of 'bar', 'pie', 'radar', 'treemap', or 'wordcloud' ▪ chartData Prop: An array of data objects. The expected object structure varies by chart type but typically includes name, count, and fill properties. ▪ setIsChartAnimating Callback: A state setter function from the parent.
.env Configuration	N/A
Error Handling	▪ A failsafe setTimeout is used to ensure the parent's animation loading state is turned off, even if a chart's onAnimationEnd callback fails to fire. ▪ The custom rendering components include logic to handle text that might otherwise overflow or render poorly, improving UI robustness.

2.4.14.2. CodedSegmentsTableModal.jsx

Purpose	Provides a comprehensive, multi-tabbed modal for analyzing coded data through three distinct views: a detailed table, dynamic visualizations, and a full statistical analysis suite.	
Dependencies	**Internal**	../theme/ThemeContext.jsx ./TableView.jsx ./VisualizationsView.jsx ./StatsView.jsx ../hooks/useStatsLogic.js ../hooks/useTableData.js
	External	react framer-motion axios react-icons

Key Components	Tabbed Interface: The core of the modal is a tabbed navigation that allows the user to switch between "Table View," "Visualizations," and "Statistical Analysis." Logic Hooks: It employs a clean architecture by delegating complex logic to custom hooks: ▪ useTableData: Manages all client-side filtering, sorting, and data aggregation for the table views. ▪ useStatsLogic: Manages the entire multi-step workflow and state for the statistical analysis tab. View Router: A renderActiveTab function conditionally renders the appropriate child component (TableView, VisualizationsView, or StatsView) based on the activeTab state. Contextual Actions: The modal header displays dynamic action buttons relevant to the active tab, such as "Export Table," "Download as PNG," or "Export as PDF." State Reset: A useEffect hook ensures the modal's state is reset to its initial configuration each time it is opened, providing a consistent user experience.
Usage	This is a top-level modal component intended to be rendered by a parent view like ProjectView. The parent controls its visibility via the show prop and provides the necessary project data (project, codeDefinitions, etc.) and export handler callbacks. The isProjectOverview prop is used to configure its title and available tabs.
Data Schema	This component orchestrates child components and hooks that interact with APIs. Its own interface is defined by its props: ▪ project Prop: The full project object. ▪ codedSegments, codeDefinitions: Arrays of the respective annotation data. ▪ handleExportToExcel, handleExportOverlaps: Callback functions to handle data exporting.
.env Configuration	This component does not directly use environment variables, but the useStatsLogic hook it consumes has an indirect dependency on VITE_BACKEND_URL for API calls.
Error Handling	▪ The component relies on the logic hooks (useStatsLogic) and parent-provided handlers to manage API errors. ▪ It manages UI states to prevent user errors, such as disabling the "Download as PNG" button while a chart is still rendering its animation.

	▪ It includes click-outside-to-close logic for dropdowns to ensure a clean UI.

2.4.14.3. D3WordCloud.jsx

Purpose	Renders a word cloud visualization by wrapping the d3-cloud library, featuring a recursive layout algorithm to ensure all words fit and a logarithmic scale for font sizes.	
Dependencies	**Internal**	N/A
	External	react d3-cloud d3-scale
Key Components	D3WordCloud: The main component that manages the word cloud generation and rendering process. Recursive Layout Algorithm: A key feature is the attemptLayout function inside the useEffect hook. If the d3-cloud library fails to place all words in the given container size, this function is called again recursively with a slightly smaller font scale. This process repeats until all words fit, making the component resilient to different container sizes and word counts Logarithmic Font Scaling: Uses d3-scale's scaleLog to map word frequencies to font sizes. This ensures that a few very frequent words don't completely dominate the visualization, allowing less frequent words to remain visible. SVG Rendering: Once the layout is calculated, the component maps over the resulting data to render each word as a styled and transformed SVG <text> element inside a main <svg> container.	
Usage	This is a presentational component. A developer should render it and provide the data array. The isDarkMode prop is used for theming, the setIsChartAnimating callback allows the parent to display a loading state during calculation, and changing the refreshKey prop will force a new layout to be generated.	
Data Schema	The component's primary data interface is its data prop: ▪ data Prop: An array of objects, where each object must have the following properties: ▪ name (String): The word to display. ▪ count (Number): The frequency of the word.	

	▪ fill (String): The hex color for the word.
.env Configuration	N/A
Error Handling	▪ The layout algorithm includes a base case to prevent an infinite loop. If it cannot fit all words even after multiple resize attempts, it logs an error to the console and stops. ▪ It gracefully handles an empty data array by rendering nothing and immediately notifying the parent that the animation is complete.

2.4.14.4. StatsView.jsx

Purpose	Serves as the main UI container for the entire statistical analysis workflow, guiding users through data summary, test selection, configuration, validation, and viewing final results.	
Dependencies	**Internal**	../stats/chi-squared/ChiSquareTypeSelector.jsx ../stats/chi-squared/ChiSquareControlPanel.js ../stats/StatsResultsPanel.jsx ../hooks/useStatsLogic.js
	External	react framer-motion axios recharts react-icons
Key Components	State Machine & View Router: The component operates as a state machine driven by a view state. A renderStatsMainContent function acts as a router, conditionally rendering different child components for each stage of the analysis process. useStatsLogic Integration: It leverages the useStatsLogic custom hook to manage all the complex state, API interactions, and business logic for the entire workflow, keeping the view component itself focused on layout and composition. Memoized Project Statistics: A useMemo hook calculates high-level summary statistics from the raw project data. This data is displayed on the initial	

	summary screen and is used to create relevant, contextual examples for the user during test selection. PDF Export Functionality: Contains a handleExportResultsAsPdf function that dynamically generates a complete HTML document with embedded print-optimized CSS, injects the rendered results, and uses the browser's print dialog to create a formatted PDF report. useImperativeHandle: Exposes the exportAsPdf function to its parent component via a ref, allowing the export to be triggered from outside the component.
Usage	This is a large, self-contained feature view. A developer would place it inside a modal or a dedicated page, providing it with the necessary project data (project, codeDefinitions, projectId) and an optional onResultsChange callback. The component then internally manages the entire multi-step user workflow.
Data Schema	This component orchestrates the useStatsLogic hook, which is the primary client for the /api/stats/run backend endpoint. Its own data interface is defined by its props: ▪ project Prop: The full project object. ▪ codeDefinitions Prop: An array of code definition objects. ▪ projectId Prop: The ID of the current project.
.env Configuration	This component does not directly use environment variables, but the useStatsLogic hook it relies on has an indirect dependency on VITE_BACKEND_URL for API calls.
Error Handling	▪ The component delegates API error handling to the useStatsLogic hook. Errors are passed down to and displayed by the StatsResultsPanel child component. ▪ The PDF export function is wrapped in a try...catch block and will alert the user if an error occurs during generation. ▪ It uses multiple confirmation modals to ensure users acknowledge important statistical assumptions before proceeding with a test.

2.4.14.5. TableView.jsx

Purpose	Renders different tabular views of coded data, including an overall summary, a by-document breakdown, and a code overlaps analysis, complete with interactive sorting and filtering controls.	
Dependencies	**Internal**	N/A

	External	react framer-motion react-icons
Key Components	View Router: The component conditionally renders one of three distinct table layouts based on the tableView prop: ▪ overall: A summary table grouping all coded segments by their code definition. ▪ byDocument: A hierarchical table grouping segments first by document, then by code. ▪ overlaps: A view that lists all text segments where multiple codes overlap, accompanied by a panel of summary statistics. Controlled Component: It is a presentational component that receives all its data and state setters as props, typically from a parent component that uses the useTableData hook. Interactive Header: Features a sticky header with a view switcher, a search bar, and an animated dropdown menu for sorting the table data by various criteria (name, frequency, etc.). Dynamic Table Rendering: The tables are dynamically generated by mapping over the pre-processed and grouped data props, using rowSpan to create clean, hierarchical layouts.	
Usage	This component is designed to be the main content of a "Table View" tab within a larger modal. A developer uses it by passing in the state and derived data from the useTableData hook.	
Data Schema	This component does not interact with APIs. Its data interface is defined by its props, which expect the structured data returned from the useTableData hook: ▪ overallGroupedData: Data grouped by code for the overall view. ▪ detailedDataByDocument: Data grouped by document, then by code. ▪ filteredOverlapsData: Data showing only instances of overlapping codes. ▪ overlapStats: An object with summary statistics for the overlaps view.	
.env Configuration	N/A	
Error Handling	▪ Empty States: The component displays user-friendly messages when there is no data to show, such as "No coded segments match your search criteria."	

	▪ UI Robustness: Includes a useEffect hook to handle clicks outside of the sort menu to close it automatically. It also has a sanitizeColor utility to prevent errors from malformed color data.

2.4.14.6. VisualizationsView.jsx

Purpose	Provides a self-contained view for the "Visualizations" tab, handling data preparation, managing the selected chart type, and rendering various charts with a selector sidebar.	
Dependencies	**Internal**	./ChartRenderer ../theme/ThemeContext.jsx
	External	react html-to-image react-icons
Key Components	Data Preparation: A useMemo hook efficiently processes the raw codedSegments prop, aggregating the data to calculate the frequency of each code and preparing it in a format suitable for the charting library. Chart Selector UI: Renders a sidebar with clickable icons that allow the user to switch between different visualization types (Bar Chart, Pie Chart, Word Cloud, etc.). ChartRenderer Integration: Renders the ChartRenderer component, passing it the prepared data and the currently selected chart type. handleDownloadChart: An async function that uses the html-to-image library to capture the current chart as a PNG file, with options for different quality levels. useImperativeHandle: Exposes the downloadChart function to the parent component via a ref, allowing the parent to trigger the download action from an external button (e.g., in a modal header).	
Usage	This is a self-contained view component. A developer should render it inside a parent layout (like a modal tab) and provide the necessary data props (codedSegments, codeDefinitions). A ref must be passed to it to enable the download functionality from the parent.	
Data Schema	The component's interface is defined by its props: ▪ codedSegments Prop: An array of coded segment objects.	

	▪ codeDefinitions Prop: An array of code definition objects, used for color mapping. ▪ setIsChartAnimating Callback: A state setter from the parent to manage a loading state while charts render. ▪ downloadChart Method: A function exposed via ref that takes a pixelRatio (Number) to control download quality.
.env Configuration	N/A
Error Handling	▪ Empty State: It displays a user-friendly "No data to visualize" message if the codedSegments prop is empty. ▪ Export Errors: The handleDownloadChart function is wrapped in a try...catch block. If the image generation fails, it logs the error and shows a browser alert() to the user.

2.4.15. theme

2.4.15.1. Logo.jsx

Purpose	Renders the application's SVG logo as a reusable and easily stylable React component.	
Dependencies	**Internal**	N/A
	External	react
Key Components	currentColor Fill: The fill attribute of the SVG paths is set to currentColor. This is a key feature that allows the logo's color to be controlled via the standard CSS color property or utility classes (e.g., Tailwind's text-blue-500). Prop Spreading: The component uses the spread operator ({...props}) to pass any props like className or style directly to the root <svg> element, making it flexible to style and size.	
Usage	This component is designed to be used anywhere the application logo is needed. A developer can import it and apply standard CSS class names to control its size and color.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	This is a static component with no internal logic. It does not have any specific error handling mechanisms.	

2.4.15.2. ThemeContext.jsx

Purpose	Creates a centralized system for managing the application's visual theme (light/dark mode), persisting the user's choice to localStorage, and providing a simple hook for components to access and modify the theme.	
Dependencies	**Internal**	N/A
	External	react
Key Components	ThemeProvider: A provider component that wraps the application. It contains the logic to read the theme from localStorage, apply the corresponding dark class to the root <html> element, and save any changes. useTheme: A custom hook that allows any child component to easily access the current theme ('light' or 'dark') and the toggleTheme function. toggleTheme(): A function exposed by the context that switches the theme between its two states.	
Usage	The <ThemeProvider> should be placed high up in the component tree, typically in main.jsx, to wrap the entire application. Any child component can then use the useTheme() hook to toggle the theme or apply theme-specific styles.	
Data Schema	The useTheme hook returns an object with the following structure: ▪ theme (String): The current theme, either 'light' or 'dark'. ▪ toggleTheme (Function): A function to switch between themes.	
.env Configuration	N/A	
Error Handling	▪ This component contains no explicit error handling. If a component attempts to use the useTheme hook outside of a ThemeProvider, React will throw an error as expected.	

2.4.15.3. ThemeToggle.jsx

Purpose	Provides a simple button for switching the application's visual theme between light and dark modes.	
Dependencies	**Internal**	./ThemeContext.jsx
	External	react

		react-icons
Key Components	useTheme Hook: The component consumes the ThemeContext via the useTheme hook to access the current theme and the toggleTheme function. Conditional Styling: It dynamically adjusts its colors based on the current theme and a navbar prop, ensuring it is always visually appropriate for its background. Conditional Icon: It displays a moon icon (MdDarkMode) in light mode and a sun icon (MdLightMode) in dark mode to intuitively indicate the action the button will perform.	
Usage	This component is designed to be placed anywhere within the application tree that is wrapped by the ThemeProvider. It works without any props, but a navbar prop can be added for specific styling when used in a dark navigation bar.	
Data Schema	N/A	
.env Configuration	N/A	
Error Handling	This component has no internal error handling. It relies on being a child of the ThemeProvider; if used outside of that context, the useTheme hook will fail.	

2.5. Supporting Files and Directories

- **public/:** The directory for static assets that can be served directly without being processed by the build tool.
- **node_modules/:** The directory where all third-party dependencies downloaded by npm install are stored. It is excluded from version control.
- **tests/**: The directory containing all automated test files for the project.
- **.Dockerignore:** Similar to .gitignore, this file specifies which files to exclude from the Docker build context to create a smaller and more secure container image.
- **.env & .env.test:** .env: A file for storing environment-specific variables, such as the backend API URL (VITE_BACKEND_URL) and links for Google Forms (VITE_GOOGLE_FORM_URL, VITE_FEEDBACK_FORM_URL). This file is kept out of version control for security.
- **.gitignore:** A file that tells the Git version control system which files and directories to ignore, such as node_modules and .env files.
- **Dockerfile:** A script containing instructions to build a portable Docker container image for the frontend application, simplifying deployment.
- **eslint.config.js:** The configuration file for ESLint, the tool used for static code analysis to enforce code quality and style rules.

- **tailwind.config.js:** The configuration file for the Tailwind CSS framework, used for defining the project's design system, including colors, spacing, and fonts.
- **vite.config.js:** The configuration file for Vite, the build tool used for the development server and production bundling.
- **package-lock.json:** An auto-generated file that records the exact version of every dependency, ensuring consistent and reproducible builds across different machines.
- **package.json:** The project's manifest file, which contains metadata, a list of dependencies, and the command scripts (dev, build, lint).

3. Statistics Microservice

This microservice is a lightweight Python application built with Flask, dedicated to performing statistical calculations. It exposes a single API endpoint that receives data, runs the appropriate statistical test, and returns the formatted results.

3.1. stats-microservice Directory

```
stats-microservice/
├── app.py
├── requirements.txt
├── Dockerfile
├── .env
├── stats-env/
├── .gitignore
└── .Dockerignore
```

3.2. Commands

This section details the primary commands used to install dependencies and run the statistical microservice. Unlike the Node.js backend, these commands are typically run directly in the terminal rather than through a package.json script section.

3.2.1. .\stats-env\Scripts\Activate.ps1

- Purpose: To activate the project's isolated Python virtual environment.
- Description: This command modifies your current terminal session to use the Python interpreter and tools contained within the stats-env directory. Activating the environment ensures that subsequent commands like python and pip are correctly scoped to this project, preventing dependency conflicts with other projects. This specific script is for use in a Windows PowerShell terminal.
- Script: .\stats-env\Scripts\Activate.ps1

3.2.2. pip install -r requirements.txt

- Purpose: To install all the required Python packages for the project.
- Description: This command uses pip, the Python package installer, to read the requirements.txt file and install all the listed dependencies (like Flask, SciPy, and NumPy) into the active Python environment. This is the first step needed to prepare the application for execution. It's recommended to run this command after activating the virtual environment (stats-env).
- Script: pip install -r requirements.txt

3.2.3. python app.py (Development)

- Purpose: To start the application in development mode.
- Description: This command runs the main application file (app.py) directly. The script is configured to start the built-in Flask development server, which automatically enables debugging and reloads the server whenever a code change is detected. The server host and port are configured via the .env file.
- Script: python app.py

3.2.4. python app.py (Production)

- Purpose: To start the application using a production-ready web server.
- Description: To run the application in production, you must first modify the app.py file. The development server block should be commented out, and the production block, which uses the
- Waitress WSGI server, should be uncommented. Waitress is a more robust and secure server suitable for handling live traffic.
- Script: python app.py (after modifying the file for production).

3.3. Stats Module

3.3.1. app.py

Purpose	Provides a dedicated microservice using Python and Flask to perform statistical calculations, primarily various Chi-Square tests and Fisher's Exact Test, for the main application.	
Dependencies	**Internal**	N/A
	External	flask flask-cors numpy scipy waitress python-dotenv
Key Components	Flask App: The main app instance that receives requests, routes them, and sends back responses. ▪ handle_test_request(): The primary route handler that acts as a router, inspecting the JSON payload to determine which statistical test to perform and calling the appropriate function.	

	Statistical Functions: ▪ run_chi_square_goodness_of_fit(): Performs the Chi-Square Goodness-of-Fit test. ▪ run_chi_square_independence(): Performs the Chi-Square Test of Independence. ▪ run_chi_square_homogeneity(): Performs the Chi-Square Test of Homogeneity. ▪ run_fishers_exact_test(): Performs Fisher's Exact Test. ▪ NaNEncoder: A custom JSON encoder that safely handles NaN (Not a Number) and Infinity values produced by statistical calculations, converting them to null for valid JSON output. Helper Functions: ▪ get_interpretation(): Generates a human-readable text summary of the test results based on the p-value. ▪ calculate_cramers_v(): Calculates the Cramér's V effect size for contingency table tests.
Usage	This microservice exposes a single, versatile endpoint. POST / ▪ Description: Executes a statistical test based on the provided JSON data. ▪ Body: (See Data Schema section for detailed structure) ▪ Success Response: 200 OK with a detailed JSON object containing the test name, subtype, statistic, p-value, degrees of freedom, interpretation, and other relevant metrics. ▪ Error Response: 400 Bad Request for invalid JSON, unsupported test types, or invalid data for calculation (e.g., empty tables). 500 Internal Server Error for unexpected calculation failures.
Data Schema	The request body for the main endpoint requires testType and subtype to route the request, along with data specific to that test. Common Fields: ▪ testType (String, required): e.g., 'chi-square'. ▪ subtype (String, required): e.g., 'goodness-of-fit', 'independence', 'fishers-exact'. For subtype: 'goodness-of-fit': ▪ observed (Array of Number, required): The observed frequencies.

	▪ distribution (Object, required): Defines the expected distribution (e.g., { "type": "uniform" }). For subtype: 'independence', 'homogeneity', or 'fishers-exact': ▪ observed (2D Array of Number, required): The contingency table. ▪ rowLabels (Array of String, required): Labels for the table rows. ▪ colLabels (Array of String, required): Labels for the table columns.
.env Configuration	FLASK_HOST: The host address the Flask server binds to (defaults to 127.0.0.1). FLASK_PORT: The port the Flask server listens on (defaults to 5001).
Error Handling	▪ Input Validation: The main route handler validates the request body for valid JSON and supported testType and subtype values, returning a 400 Bad Request on failure. ▪ Data Validation: Each statistical function checks its input data for validity (e.g., non-empty arrays, correct dimensions, non-zero totals) and raises a ValueError that results in a 400 Bad Request with a descriptive message. ▪ Calculation Errors: A global try...except block catches any unexpected exceptions during computation and returns a 500 Internal Server Error. ▪ Serialization: Uses a custom NaNEncoder to prevent crashes when serializing statistical results that may contain NaN or Infinity, ensuring a valid JSON response is always sent.

3.4. Supporting Files and Directories

- **requirements.txt:** This file lists all the Python packages required for the microservice to run, including flask, scipy, and numpy. It ensures a consistent and reproducible setup of the environment.
- **Dockerfile:** This is a script containing instructions to build a portable Docker container image for the application. It defines the base Python environment, copies the application code, and installs the required dependencies from requirements.txt.
- **.env:** This file stores environment variables for the application, such as the FLASK_HOST and FLASK_PORT. This separates configuration from the source code, allowing for different settings in development and production.
- **stats-env/:** This directory is the Python virtual environment. It creates an isolated workspace for the project's specific dependencies, preventing conflicts with other Python projects. It contains a copy of the Python interpreter and the libraries installed from requirements.txt. This directory is typically excluded from version control.
- **.gitignore**: A configuration file for the Git version control system that specifies which files and directories to ignore (e.g., node_modules/, .env).

- **.Dockerignore**: Specifies which files should be excluded from the Docker container to ensure a lean and secure build.