**LeBeDigital** - Web App Documentation

Test server: <https://test.lebedigital.bam.de>  
Git: [BAMresearch/LebeDigital at lebedigital-v1.1.0](https://github.com/BAMresearch/LebeDigital/tree/lebedigital-v1.1.0)

### ****1. Introduction****

The **LeBeDigital** project, part of the **Plattform Material Digital** initiative, aims to establish a robust structure for managing concrete material data. By adhering to the FAIR principles (Findable, Accessible, Interoperable, Reusable), the project provides a flexible framework to accommodate future advancements in concrete research and production. The core deliverable is a **material database** that integrates concrete-specific characteristic values and models into a structured system.

#### ****Key Features****

* **Raw Data Upload**
* **Metadata Extraction**
* **Knowledge Graph Mapping**
* **Data Retrieval**

### ****2. Getting Started****

#### ****2.1 Prerequisites****

To set up and run the project, ensure the following are installed:

* **Python**: Version 3.12.2 or higher (other versions may work but are untested).
* **Git**: For cloning the repository.
* **Visual Studio Code (VS Code)**: Recommended IDE for development and debugging.

#### ****2.2 Installation****

1. **Clone the Project**  
   Open your terminal and clone the repository:

git clone https://github.com/BAMresearch/LebeDigital.git

1. **Switch to the staging Branch**  
   The staging branch contains the latest updates. Navigate to it using:

git checkout staging

Ensure you are on the correct branch by running:

git branch

The **staging** branch should be highlighted.

1. **Set Up a Virtual Environment**  
   To isolate project dependencies, use a Python virtual environment:
   * Create the virtual environment:

python -m venv venv

* + Activate the virtual environment:
    - On Windows:

venv\Scripts\activate

* + - On macOS/Linux:

source venv/bin/activate

1. **Navigate to the Web Directory**  
   Move into the web folder, where the main application files are located:

cd LebeDigital/web

1. **Install Dependencies**  
   With the virtual environment activated, install the required Python libraries:

pip install -r requirements.txt

1. **Set Up Configuration**
   * Copy the configuration template to a working configuration file:

cp config.template.json config.json

* + Open the config.json file and update the placeholders with your **Fuseki** credentials:
    - **Username**: Your BAM username.
    - **Password**: Your BAM password.  
      These credentials are used to connect to the Fuseki triple store at:  
      [**http://fuseki.bam.de**](http://fuseki.bam.de).

1. **Run the Application**  
   Start the server:

python server.py

1. **Access the Application**  
   Open a browser and navigate to http://localhost:5000 to view the application.

### ****3. Git Workflow****

To maintain a clean and efficient development process, we follow a structured Git workflow designed to minimize conflicts and ensure that the staging branch remains updated with the latest stable features. Here’s how we manage our branches and code contributions:

#### ****3.1 Feature Development****

1. **Create a Feature Branch**  
   When starting work on a new feature, create a new branch from the staging branch:

git checkout staging

git pull # Ensure you have the latest updates from staging

git checkout -b feature-branch-name

1. **Work on the Feature**  
   Make changes and commit them to your feature branch:

git add .

git commit -m "Describe the changes you made"

git push

1. **Update Your Feature Branch Before Merging**  
   Before merging your changes back to staging, ensure your feature branch is up to date.
2. **Merge to Staging**  
   After resolving any conflicts (if applicable) and testing your feature, create a pull request from your branch to staging, and merge it.
3. **Delete the Feature Branch**  
   Once the feature is successfully merged and tested, delete the feature branch from GitHub.

#### ****3.2 Deployment and Versioning****

To manage deployment, we use versioning to indicate the type of changes being introduced. The current version is **1.1.0**, with the numbering scheme as follows:

* **Major (1.x.x)**: For significant changes or overhauls to the application.
* **Minor (x.1.x)**: For new features or minor updates.
* **Patch (x.x.1)**: For bug fixes or minor tweaks.

1. **Creating a Deployment Branch**  
   When preparing for deployment, create a new branch from the latest running version:

git checkout lebedigital-v1.1.0 # the current production branch

git checkout -b lebedigital-v1.2.0

1. **Pull Latest Changes**  
   Pull the latest staging updates into the newly created deployment branch:

git pull origin staging

1. **Test the New Branch**
   * Perform **unit testing** and **integration testing** to ensure all features work as expected. Currently we do the testing manually.
2. **Deployment**  
   Once testing is complete and the branch is stable, notify the IT team to deploy the new branch on the server. Then test the application on <https://test.lebedigital.bam.de> to verify everything functions properly.

### ****4. Project Structure****

To provide clarity on how the project is organized, here’s an overview of the main directories and files:

* **Root Directory**
  + Contains knowledge graph templates (/cpto), necessary scripts, documentation, and essential files like .gitignore, README.md, LICENSE etc.
* **web/**
  + Core application folder containing all web app related files.
* **Subdirectories**
  + **templates/**: HTML templates rendered by Flask. Below is a detailed description of each template:
    - **welcome.html:** The landing page of the application, typically shown when the user first visits the site.
    - **base.html**: The base layout file that includes the structure for the website’s common elements, such as the navigation bar and side menu. Other pages will extend this template for consistent layout across the app.
    - **database.html:** Displays the Database page.
    - **queryPage.html**: SPARQL query page.
    - **uploadForm.html:** Upload Data page.
    - **newMixture.html:** Input form for adding mixture.
    - **newCompressiveStrength.html:** Input form for adding compressive strength.
    - **newEmodule.html:** Input form for adding E-module.
    - **signup.html:** The registration page where users can create an account.
    - **login.html:** The login page where users can enter their credentials to access the application.
    - **myFiles.html:** A page to display the uploaded files by the user.
    - **admin.html:** An administrative page for managing users, viewing logs, configurations, or clearing datasets. This page is typically restricted to admin users for system maintenance.
  + **static/**: This folder contains static files like CSS, JavaScript, and images. These files are linked in the HTML templates. The files used across all templates are linked in the base.html file, while files specific to certain pages are linked directly from the relevant templates.
  + **logs/**: log files for debugging the app.
* **Key Files**
  + **requirements.txt**: List of Python dependencies.
  + **config.template.json**: Configuration template file. You need to create config.json file by copying this, and replacing the placeholders.
  + **server.py**: Main application entry point.

### ****5. server.py Overview****

The **server.py** file is the main entry point of the application. It handles the following responsibilities:

* **Flask Application Initialization**:
  + Initializes the Flask application and sets up various configurations for the app, such as secret keys, session handling, and database connections.
* **Configuration Setup**:
  + The file checks for the existence of a config.json file. If it does not exist, it creates one based on the config.template.json. The SECRET\_KEY is either loaded or generated for securing sessions.
* **Backend Logic & Routes**:
  + **Handles all routes and business logic**: All the application's routes (URLs) and the logic behind them are defined in server.py. It is responsible for receiving HTTP requests, processing data, and sending responses.
  + Each route in Flask corresponds to a specific URL and HTTP method (GET, POST, etc.), and these routes are defined using Flask’s decorators, such as @app.route().
* **Error Handling and Logging**:
  + Configures logging to capture different levels of information (ERROR, INFO, DEBUG) and stores logs in a logs/ directory. The logs include important information like errors and server activity for debugging and monitoring purposes.

#### ****6. Database Overview****

The application uses two main databases: users.db, main.db

During the first run, server.py initializes the following:

* **User Database**: A user table is created using SQLAlchemy for managing user credentials securely.
* **Main Database**: An SQLite database (main.db) is created with the following tables:
  + **uploads**: Stores metadata about uploaded files, user information, file types, and mappings.
  + **uidlookup**: Associates unique IDs with names for efficient lookups.

##### **Notes**

* The .db files are automatically created in the application's root directory if they do not exist.
* Logs for database connection errors are recorded in the logs directory.

## **7. Web App Pages**

### ****7.1 Data Management****

#### ****7.1.1**** [Database](https://test.lebedigital.bam.de/database) Page

**Features**

* **View Uploaded Data**: Displays a list of uploaded files, showing their ID, Name, and Type.
* **View Details**: Clicking on any entry displays additional details, including item names, values, and units.
* **Search**: Allows users to search for specific files, projects, or data entries.
* **Download Options**: Users can download the displayed table as a CSV or raw data for individual files.

**Workflow**

1. **Initialization**  
   The page is initialized in server.py via the /database route:

@app.route('/database', methods=['GET', 'POST'])

1. **Frontend Table Handling**

* The table is managed by **Tabulator.js** and rendered dynamically using logic defined in database-page.js.
* The table supports pagination, sorting, and searching for a better user experience.

1. **User Interactions**

* **View Details**: Clicking on an individual file calls executeSparqlQuery to retrieve related data.
* **Unit Links**: Clicking on a unit value opens its details in a new tab (e.g., linking to qudt.org).
* **Download Raw Data**: A "Download" action retrieves and downloads the raw file associated with a specific entry. This is managed via downloadRawData.

1. **Data Transformation**

* The table columns (generateColumns) and data rows (transformData) are dynamically generated to adapt to executeSparqlQuery responses.
* Units are appended to corresponding values, ensuring clarity in the displayed data.

1. **Additional Features**

* **Table Download**: Users can export the table to a CSV file using the downloadTable function.
* **Back Navigation**: The "Back" button restores the previous table view (goBack method).
* **Dynamic Table Creation**: Tables are changed based on user interactions via the create\_query method.

**Overview**  
To explore or customize specific functionality:

* The backend logic is in server.py (/database route).
* The frontend interactions are in database-page.js, with key methods:
  + executeSparqlQuery: Gets data for individual file.
  + create\_query: Constructs and sends queries dynamically.
  + filterTableData: Filters table data based on user input.
  + downloadRawData: Manages raw data downloads for individual files.

#### ****7.1.2**** [SPARQL Query](https://test.lebedigital.bam.de/query)

#### Features:

#### SPARQL Query Execution: Execute SPARQL queries against the backend triple store (Fuseki).

#### Example Queries: Select predefined queries without writing it.

#### Dynamic Table Updates: Display query results in a tabular format with options for sorting, filtering, and pagination using tabulator.js library.

#### Data Visualization: supports dynamic chart generation based on query results using chart.js library. Charts include bar, scatter plots, and more, depending on the query type.

#### Download Data: Export table data as CSV for offline analysis.

**Workflow**

1. **Initialization**  
   The page is initialized in server.py via the /query route:

@app.route('/query')

#### 2. ****Query Execution and Visualization****

**Function:** runQuery()

* Retrieves the user's query and passes it to executeSparqlQuery().
* Updates the results table and chart dynamically based on the response.

#### 3. ****Data Table****

**Function:** initializeOrUpdateTable(data)

* Initializes or updates a Tabulator-based table to display query results.
* Includes pagination, filtering, and sorting features.

#### 4. ****Chart Visualization****

**Function:** updateChart(data, vars)

* Dynamically determines chart type based on query variables.
* Supports multiple chart types configured in queryVisualizationSettings.

#### 5. ****Download Table****

**Function:** downloadTable()

* Allows users to export the current table view as a CSV file.

**Overview**  
To explore or customize specific functionality:

* The backend logic is in server.py (/query route).
* The frontend interactions are in querypage.js, with key methods:
  + **executeSparqlQuery(query)**  
    Sends the SPARQL query to the backend using a POST request. Returns the result data in JSON format.
  + **initializeOrUpdateTable(data)**  
    Creates or refreshes the Tabulator table using dynamic columns based on query variables.
  + **updateChart(data, vars)**  
    Determines the appropriate chart type (bar, scatter, etc.) and updates the chart with query results.
  + **determineQueryType(vars)**  
    Determines the visualization type based on query variables.
  + **Object:** queries  
    A set of predefined SPARQL queries that can be inserted as example-query.

#### 7.1.3 [Upload Data](https://test.lebedigital.bam.de/upload)

#### ****Features****

1. **Upload Mixture/Compressive Strength/E-Module**
   * Files can be uploaded directly from a device, via URL, or through manual input.
   * To upload any data other than Mixture, the user must first select a Mixture.
   * Supported file formats:
     + **Mixture**: .xlsx, .xls, .json
     + **Compressive Strength (CompSt)**: .dat, .xml, .json
     + **E-Module**: .xml, .json
2. **Use Previous Mixture**
   * Access existing Mixtures via a searchable dropdown.
   * Select a pre-existing Mixture to upload related data such as CompSt or E-Module.
3. **Upload Multiple Files**
   * Supports uploading multiple files simultaneously for Compressive Strength and E-Module.
4. **Upload from URL**
   * Upload files directly from public URLs hosted on platforms like GitHub, Google Drive, or Dropbox.
   * Ensure URLs are publicly accessible for successful uploads.

#### ****Workflow****

##### **1. Initialization**

* The page is initialized in server.py using the /upload route:

@app.route('/upload')

##### **2. Frontend Visualization and Logic**

* **HTML Template**: uploadForm.html
* **JavaScript Files**:
  + main.js
  + getMixtures.js

###### getMixtures.js

* Loads Mixture data from the server and populates the dropdown menu.
* Checks the URL for a pre-selected Mixture (mixtureId parameter).
* Uses **Select2** to enhance the dropdown UI with search and style capabilities.
* Automatically selects a Mixture if specified in the URL.
* Handles errors during data loading and displays appropriate feedback.

###### main.js

**Key Functionalities:**

* **Global Variables:**
  + mixtureID: Stores the current Mixture ID.
  + mixtureName: Stores the current Mixture name.
  + institute: Set to 'BAM'.
* **Button Controls:**
  + enableUploadButton(): Enables all upload buttons.
  + disableUploadButton(): Disables all upload buttons.
* **Mixture Management:**
  + showMixtureName(): Displays the selected Mixture name in the UI.
  + removeMixtureId(): Clears the selected Mixture ID and name from the UI.
  + generateUUIDv4(): Generates a unique identifier for new Mixtures.
* **Upload Functionality:**
  + uploadData(type, fileID, urlID, label):
    - Handles file/URL uploads based on the type.
    - Creates a new Mixture ID if the type is "Mixture."
    - Sends data to the server using FormData.
    - Displays success/error notifications.
* **File Handling:**
  + clearFileInput(fileID): Clears the file input field and disables the upload button.
  + onFileSelected(event, fileLabel):
    - Validates file formats.
    - Handles multiple file selections.
    - Supports various file types for Mixture, Compressive Strength, and E-Module.
* **URL Management:**
  + onUrlEntered(urlInput, fileLabel): Validates and processes entered URLs.
  + isValidURL(string): Confirms if a string is a valid URL.
* **Navigation Functions:**
  + toggleSections(): Switches between sections for existing and new Mixtures.
  + goToMixtureUpload(): Scrolls to the Mixture upload section.
  + goToMixtureForm(): Redirects to the new Mixture form.
  + checkMixtureAndRedirect(targetPage): Verifies Mixture selection before navigation.
  + redirectToPage(page): Adds the Mixture ID to the URL and navigates to the specified page.
  + goToComStForm(): Redirects to the Compressive Strength form.
  + goToEModuleForm(): Redirects to the E-Module form.
* **Event Listeners:**
  + Mixture selection change handler: Updates mixtureID and mixtureName when the dropdown selection changes.

##### **3. Backend Functions**

1. **File Handling:**
   * Uploaded files are processed through the following endpoint:

@app.route('/dataUpload', methods=['POST'])

* + The endpoint first checks if the file already exists in the database (uploads table).

1. **Data Mapping and Conversion:**
   * If the file does not exist, it is inserted into the uploads table.
   * The file is then processed using async\_function, which:
     + Runs necessary mapping scripts based on the file type.
     + Converts the uploaded file to TTL format.
     + Updates the database and inserts the TTL into Fuseki.

#### 7.2. Input Forms

#### 7.2.1 [Mixture](https://test.lebedigital.bam.de/new_mixture)

#### User can input mixture manually via this form. The required fields are Lab, Mixing Date, and Human-Readable Id. All other items can have one or multiple inputs.

#### ****Workflow****

##### **1. Initialization**

* The page is initialized in server.py using the /new\_mixture route:

@app.route('/new\_mixture)

##### **2. Frontend Setup**

* **HTML Template**: newMixture.html
* **Script:**
  + createDynamicInput
    - Creates new input rows for components
    - Clones existing input template
    - Clears values in new inputs
  + updateNames
    - Updates field names dynamically
    - Formats names in a specific format using regular expression
    - Maintains proper indexing for form submission
  + File Input: Validates file size (2MB limit)
  + Form Submit
    - Validates form data
    - Creates FormData object
    - Handles file attachments
    - Submit to server.py > submit\_mixture
  + Error Handling: Checks for duplicate IDs

**3. Backend Functions**

submit\_mixture() function:

* Handles mixture submission to database
* Processes form data and files
* Performs authentication check
* **Data Processing** 
  + Converts form data to dictionary
  + Filters out empty values
  + Adds standard unit definitions
* **Data Organization**
  + Mixture components:
    - Required fields (Lab, MixingDate, HumanReadableID)
    - Binder
    - Water
    - Aggregate
    - Addition
    - Admixture components
    - Fiber components
    - Air components
* **Database Operations**
  + Checks for duplicate Human-readable IDs
  + Inserts mixture data into 'uploads' table with: User information, Mix JSON data, Additional file data etc.
  + Calls async\_function for mapping

#### 7.2.2 [Compressive Strength](https://test.lebedigital.bam.de/new_compressive_strength)

#### User can input Compressive Strength manually via this form. All the fields are required. User must select one Mixture to proceed.

#### ****Workflow****

##### **1. Initialization**

* The page is initialized in server.py using the /new\_compressive\_strength route:

@app.route('/new\_compressive\_strength)

##### **2. Frontend Setup**

* **HTML Template**: newCompressiveStrength.html
* **Scripts:**
  + getMixtures.js: To generate mixture dropdown
  + specimenShape.js: For different specimen shapes, it loads different fields.
  + File Input: Validates file size (2MB limit)
  + Form Submit
    - Validates form data
    - Creates FormData object
    - Handles file attachments
    - Submit to to server.py > submit\_compressive\_strength
  + Error Handling: Checks for duplicate IDs

**3. Backend Functions**

submit\_compressive\_strength() function:

#### Handles submission of compressive strength test data

#### Processes two JSON objects (comst and specimen data)

#### Manages additional file uploads

#### Database Operations

#### Checks for duplicate Human-readable IDs

#### Inserts into 'uploads' table:

#### User information

#### Both JSON blobs (comst and specimen)

#### Additional file data

#### Mixture ID reference

#### Mapping and TTL file generation: calls async\_function with UniqueID

#### 7.2.3 [E-Module](https://test.lebedigital.bam.de/new_emodule)

#### User can input E-Module data manually via this form. All the fields are required except measured Compressive Strength. User must select one Mixture to proceed.

#### ****Workflow****

##### **1. Initialization**

* The page is initialized in server.py using the /new\_emodule route:

@app.route('/new\_emodule)

##### **2. Frontend Setup**

* **HTML Template**: newEmodule.html
* **Scripts:**
  + getMixtures.js: To generate mixture dropdown
  + specimenShape.js: For different specimen shapes, it loads different fields.
  + File Input: Validates file size (2MB limit)
  + Form Submit
    - Validates form data
    - Creates FormData object
    - Handles file attachments
    - Submit to server.py > submit\_emodule
  + Error Handling: Checks for duplicate IDs

**3. Backend Functions**

submit\_emodule() function:

#### Handles submission of compressive strength test data

#### Processes two JSON objects (comst and specimen data)

#### Manages additional file uploads

#### Database Operations

#### Checks for duplicate Human-readable IDs

#### Inserts into 'uploads' table:

#### User information

#### Both JSON blobs (emodule and specimen)

#### Additional file data

#### Mixture ID reference

#### Mapping and TTL file generation: calls async\_function with UniqueID

#### 7.3. Others

#### 7.3.1 [My Files/All Files](https://test.lebedigital.bam.de/files)

#### User can see all the files s/he uploaded from this page, also download, or delete any file. For the Admin, the page name is “All Files”, since admin can see the files with the upload info (e.g. user who uploaded it). When a user deletes a file, the admin sees a delete tag beside that file. Only the admin can permanently delete any file from the database.

#### ****Workflow****

##### **1. Initialization**

* The page is initialized in server.py using the /files route:

@app.route('/files)

##### **2. Frontend**

* **HTML Template**: myFiles.html

**3. Functions**

* **downloadFile(id): Downloads a file using its unique ID**
  + **Makes GET request to rawDownload URL with file ID**
  + **Converts response to blob**
  + **Creates temporary download link**
  + **Triggers automatic download**
* **deleteFile(id): Deletes a file with user confirmation**
  + **Shows confirmation dialog**
  + **Gets user type (admin/regular)**
  + **Selects appropriate endpoint:**
    - **Admin: get\_admin\_data**
    - **others: update\_deleted\_by\_user**
  + **Sends DELETE request with file ID**
  + **If request is sent by admin, the file is deleted permanently, else soft delete is performed by updating the delete column to 1 in the uploads table.**
  + **ISSUE: file is only deleted from DB, but it needs to be also deleted from the triple store (Fuseki), so that it remains consistent and does not appear in the SPARQL Query result.**

#### 7.3.2 [Settings](https://test.lebedigital.bam.de/admin)

#### Only the admin can access this page. The feaures of this page are: View or delete logs (info, debug, error), view the list of users, view configuration setup, and delete all files, both from database and triple store (Fuseki).

### ****1. Initialization****

* The page is initialized in server.py using the /admin route:

@app.route('/admin)

### ****2. Frontend****

* **HTML Template**: admin.html

### **3.** Functions

### displayConfig(config): Displays configuration settings in a table format

### displayUsers(users): Shows list of system users

### displayLogs(logs)

### Creates an interactive table of system logs

### Displays filename, size, and last modified date

### Pagination with customizable page sizes

### Sortable and serachable columns

### Visual indicators for log types (i.e. info, debug, error)

### getLogType(filename): Determines the category of a log file

### getLogTypeColor(type): Assigns color to log types

### showLogContent(logData): Displays detailed log content in a modal

### clearData(): Removes all content from database and Fuseki server

### clearLogs(): Deletes all system log files

#### 