LETICIA 1.0 - User Manual

simuLatEd Task and actIvity Capture Interactive plAtform

Document version: v1.0.2
October 2022

Author:
Daniel Gacitúa (daniel.gacitua@usach.cl)

Table of Contents

Introduction

LETICIA is an open-source and customizable web platform for interactive information retrieval (IIR) studies. Its main purpose is to serve as a base for user study design and implementation in IIR, with little to no coding requirements to create a study.

LETICIA is built as an effort to ease study creation for IIR researchers and offer a state of the art platform to be applied both in local and remote environments. LETICIA can be used as-is from its already developed modules, or as a starting point for developing custom modules and functionalities.

Features

- Fully operational platform for search tasks in IIR studies
- User interaction capture (visited pages, queries, keyboard and mouse actions, among others)
- Simulated web search environment (powered by Apache Solr)
- Non-relational database storage (powered by MongoDB)
- Easy deploy server (on GNU/Linux)
- No install required for participants (runs through web browser)
- Integrated out of the box web crawler
- Support for search task personalization
- Customizable study experience (for participants)
- Customizable forms, questionnaires and studies
- Multi-language frontend (i18n support)
- Extensible and modular approach (through its REST API)
- Released as open source software (AGPL v3)

Install and configure LETICIA

This platform must be installed on a server machine and can be served to participants (clients) directly from the browser (no installation required).

Please bear in mind that to install LETICIA you will need to have at least basic knowledge of the use of the terminal/console on *nix environments. LETICIA was designed to be installed on Linux servers, but it might also work on Windows/macOS environments (not tested).

To install LETICIA, you have three options: Dev mode, Native mode, and Docker mode.

Hardware requirements

To deploy LETICIA's web platform, a server with at least 1GB RAM and one dedicated CPU core is required. Also, an internet connection to the server is required for dependency downloading and connection from participants. LETICIA can be deployed on a Virtual Private Server (VPS) service like DigitalOcean, Amazon Web Services, or Google Cloud Platform.

Participants on the LETICIA platform can use Windows, macOS, or Linux as their operating system. The only requirement is to use an evergreen web browser (like Google Chrome, Mozilla Firefox, or Microsoft Edge) with an internet connection to access LETICIA's web platform.

Environment variables

In any of these three deployment modes (Dev, Native, or Docker), you will need to set environment variables for LETICIA to work. A default example of these variables is found on `src/.env.example`, you need to copy, customize, and rename this file to `src/.env` to set custom deploy behaviors.

The following environment variables are currently available on this version of LETICIA:

Env Variable	Default Value	Description
LETICIA_HOST	localhost	IP address or DNS domain where LETICIA is running
LETICIA_PROTOCOL	http	Protocol used by LETICIA (http or https)
LETICIA_PORT	3000	Local port to deploy LETICIA's WebApp
MONGODB_HOST	localhost	URL for MongoDB
MONGODB_PORT	27017	Port for MongoDB

MONGODB_DATA_DB	leticia-data	Database name for experimental data
MONGODB_USER_DB	leticia-user	Database name for credential data
SOLR_HOST	localhost	URL for Solr (inverted index)
SOLR_PORT	8983	Port for Solr
SOLR_CORE	leticia	Core/collection for Solr
ENABLE_GOOGLE_LOGIN	true	Toggles Google SSO Login
GOOGLE_CLIENT_ID	insert-id	Google SSO Login client id
GOOGLE_CLIENT_SECRET	insert-secret	Google SSO Login client secret
ENABLE_FACEBOOK_LOGIN	true	Toggles Facebook SSO Login
FACEBOOK_CLIENT_ID	insert-id	Facebook SSO Login client id
FACEBOOK_CLIENT_SECRET	insert-secret	Facebook SSO Login client secret
ENABLE_EMAIL_LOGIN	true	Toggles Email Login
JWT_KEY	secret	String for JWT token generation
LETICIA_PILOT_MODE	false	Toggles LETICIA's Pilot Mode (deprecated)
CURRENT_SESSION_FLOW	short	Activity flow to run with new participants
ENABLE_FRONTEND	true	Toggles deploy of LETICIA's Frontend
ENABLE_API_DOCS	true	Toggles deploy of LETICIA's OpenAPI Documentation

Dev mode install

This mode is intended for development purposes, it allows to modify different elements of the platform with watch mode both in frontend and backend (all changes are applied when saving any edited file).

To deploy in this mode, a Windows/macOS/Linux machine with the following installed dependencies is required:

- Node.js (tested on latest LTS v16.x)
- MongoDB (tested on version 4.4)
- Solr (tested on version 8.7.0)
- Java JDK (tested on OpenJDK 11)

Install required dependencies

The following instructions are designed for Ubuntu 20.04 LTS. Adapt them if you are using another operating system.

Node.js

Run these commands to install latest Node.js LTS:

```
$ curl -sL https://deb.nodesource.com/setup_lts.x | sudo -E bash -
$ sudo apt-get install -y nodejs
```

MongoDB

Run these commands to install MongoDB:

```
$ wget -qO - https://www.mongodb.org/static/pgp/server-4.4.asc | sudo apt-key add -
$ echo "deb [ arch=amd64,arm64 ] https://repo.mongodb.org/apt/ubuntu focal/mongodb-org/4.4
multiverse" | sudo tee /etc/apt/sources.list.d/mongodb-org-4.4.list
$ sudo apt-get update
$ sudo apt-get install -y mongodb-org
$ sudo systemctl start mongod
$ sudo systemctl enable mongod
```

Java and Solr

Solr requires Java 8 JDK or greater to run. Run these commands to install OpenJDK:

```
$ sudo apt-get update
$ sudo apt-get install default-jre default-jdk
```

Run the following commands to install Solr:

```
$ cd /tmp
$ wget https://archive.apache.org/dist/lucene/solr/8.7.0/solr-8.7.0.tgz
$ tar xzf solr-8.7.0.tgz solr-8.7.0/bin/install_solr_service.sh --strip-components=2
```

```
$ sudo bash ./install_solr_service.sh solr-8.7.0.tgz
```

Create a new core (collection) in Solr for LETICIA:

```
$ sudo su - solr -c "/opt/solr/bin/solr create -c leticia -n data_driven_schema_configs"
```

Install LETICIA (dev)

1. Install the required dependencies (instructions are available above)
2. Download or clone this repository
3. On LETICIA's repository `src/` directory, copy `.env.example`, rename it as `.env` and edit the file to customize Environment Variables (if needed)
4. On `src/` directory, run `npm install` to install Node.js dependencies
5. On `src/` directory, run `npm run dev` to start LETICIA in watch mode for development

After running LETICIA in Dev mode, you can perform changes and they are immediately loaded on file save. To exit Dev mode, you must close the terminal/console where LETICIA is running.

Native mode install

This mode is intended to deploy the platform (or a modified version of it) optimized for use in experimental studies. With this mode, you have full control of the deployed environment through the PM2 ecosystem file.

Install required dependencies

See “Install required dependencies” for Dev mode; the Native production mode uses the same dependencies and can be installed on Linux, Windows, or macOS servers.

Install LETICIA (native)

1. Install Node.js, MongoDB, and Solr (instructions are available above)
2. Install PM2 globally for Node.js: `npm install -g pm2`
3. Download or clone LETICIA’s git repository
4. On LETICIA's repository `src/` directory, copy `.env.example`, rename it as `.env` and edit the file to customize Environment Variables (if needed)
5. On `src/` directory, run `npm install` and then `run npm run clean && npm run build`
6. On `src/` directory, run LETICIA with PM2: `pm2 start ecosystem.config.js`, other actions that can be performed with PM2 to control the platform on production are available on the PM2 documentation

Docker mode install

This mode is intended for use for quick-and-easy deployment on experimental setups. It requires fewer steps to deploy (compared to Native mode), automatically downloads all dependencies as Docker containers, and all the LETICIA environment is isolated from the server filesystem. The main drawback is that it may require more system resources than the Native mode. You will also need access to a Bash console to run the build, start, and stop scripts.

Install required dependencies

This mode only requires Docker and Docker Compose as dependencies. You can either install Docker Desktop (for Windows/macOS) or install Docker for terminal/console (for Linux) as described below.

Docker and Docker Compose

Install required dependencies for Docker

```
$ sudo apt-get update
$ sudo apt-get install ca-certificates curl gnupg lsb-release
```

Add Docker's GPG key

```
$ sudo mkdir -p /etc/apt/keyrings
$ curl -fsSL https://download.docker.com/linux/ubuntu/gpg | sudo gpg --dearmor -o
/etc/apt/keyrings/docker.gpg
```

Set up Docker's repository

```
$ echo \
   "deb [arch=$(dpkg --print-architecture) signed-by=/etc/apt/keyrings/docker.gpg] \
   https://download.docker.com/linux/ubuntu \
   $(lsb_release -cs) stable" | sudo tee /etc/apt/sources.list.d/docker.list > /dev/null
```

Install Docker and Docker Compose

```
$ sudo apt-get update
$ sudo apt-get install docker-ce docker-ce-cli containerd.io docker-compose-plugin
```

Enable using Docker for the current user (so `sudo` isn't required for Docker commands):

```
$ sudo usermod -aG docker $(whoami)
```

Then restart the current terminal/console to apply changes.

Install LETICIA (Docker)

1. Install Docker and Docker Compose (instructions are available above)
2. Download or clone LETICIA's repository

3. On LETICIA's repository `src/` directory, copy `.env.example`, rename it as `.env` and edit the file to customize Environment Variables (if needed)
4. On `src/` directory, run `./leticia-build.sh` to download and generate all required dependencies
5. On `src/` directory, run `./leticia-up.sh` to deploy LETICIA on background
6. On `src/` directory, run `./leticia-down.sh` to stop running LETICIA

Next steps

After installing LETICIA, you can either deploy an experimental study with the available components or develop your own ones to make a custom study. The section “Building an experimental flow” offers insights for building a study, while “LETICIA’s Architecture” offers an overview of the architecture, dependencies, and file layout to develop new components for LETICIA. Also, the section “Important considerations” offers tips for both development and deployment phases.

Building an experimental flow

Once you have installed LETICIA on your server, you can start designing the required components for running a simulated search task. LETICIA has four main component types for building the simulation: Stages, Challenges, Flows, and Experiment Assets. These components can be mixed to create the search simulation through the REST API available in LETICIA.

LETICIA's REST API can be accessed directly using a REST client (like curl or Postman) or through the interactive documentation powered by OpenAPI. The default URL for the REST API is `http://localhost:3000/v1/`. You can access the interactive documentation through a web browser in the default URL `http://localhost:3001/openapi/`. All bodies for requests to the REST API must follow the JSON format. The following sections describe how to generate components to build a simulated search task on LETICIA.

Stages as purposed activities

A stage is a single-purpose activity, like filling out a questionnaire, completing a test, or performing search queries on the search engine. Every stage in LETICIA is represented by a frontend template. The following stages are currently available on LETICIA:

- **Authentication (Login & Register):** To authenticate in the platform, two methods are available: User credentials (email and password) and OAuth single sign-on (either with a Google or Facebook account). Registering on the platform allows LETICIA to track relevant user actions on the database for analysis.
  Frontend path: `/oauth`
- **Informed Consent:** This stage displays a PDF document to the participant that he/she must agree to participate in the study. The PDF path location is `src/client/assets/consent.pdf` and can be replaced with a custom one.
  Frontend path: `/consent`
- **Demographic Survey:** This stage template displays a questionnaire to capture demographic information (like sex, age, academic level, and others) from the participant. The demographic survey template is located on `src/client/templates/questionnaires/Demographic.vue` for customization.
  Frontend path: `/demographic`
- **Typing Test:** This stage template can display several text snippets that must be typed by the user. The purpose is to get the typing speed and rhythm of the participant over a fixed text. Typing test snippets' format example can be seen on `src/client/assets/typingTestSamples.json` and the template import of this file is on `src/client/templates/questionnaires/TypingTest.vue`
  Frontend paths: `/typing-instructions` and `/typing`
- **Query Planning:** This stage template allows search query formulation without actually performing the search. This can be useful for planning search queries while participants' interactions are registered.
  Frontend path: `/query`
- **Task Form:** This stage template enables the building of custom questionnaires from the database. Questionnaires can be built as JSON strings and loaded to the platform using the

REST API.
Frontend path: `/taskform`

- **Search:** This module emulates the experience of a web search engine, performing the search on a limited dataset of web documents being queried from an inverted index. The participant can bookmark one or more documents as relevant for the search task.
  Frontend paths: `/search-instructions` and `/search`
- **NASA-TLX Test:** This module implements the raw NASA-TLX questionnaire (Hart et al.,1988; Byers et al., 1989) to measure perceived workload when completing a task.
  Frontend path: `/nasa-tlx`
- **Stroop Test:** This module implements a simplified version of the Stroop effect test (Stroop, 1935) to measure selective attention and cognitive load when completing a task (Gwizdka, 2008). In this version of the Stroop effect test, the participant must indicate if there is a match (or no match) between the written name of a color (like red, blue, or green) and the color it is printed on.
  Frontend path: `/stroop`

Screenshots of these stages can be seen on the Appendix.

Challenges as a time-constrained group of stages

A "challenge" is a group of stages with a specified time limit. When the timeout is reached, LETICIA forwards to the next stage/challenge. LETICIA currently has the "short challenge" and the "extended challenge" as references. The "short challenge" uses the Query Planning stage as the main activity, while the "extended challenge" uses the Search stage as the main activity. More challenges can be built from the `routes` module definition in the front end (located in `src/client/modules/routes.js`). The frontend paths for the short and extended challenges are `/short-challenge` and `/extended-challenge` respectively.

Flows as a linear collection of stages and challenges

A "flow" is a series of stages and/or challenges ordered linearly. This means there is only one possible sequence of actions for each flow (no branching paths). Flows must be populated with experiment assets to work (described in the following section).

Experiment Assets

To create an experimental “flow”, you need to load experiment assets to the LETICIA platform. These assets can be created and loaded into LETICIA through its REST API using the JSON format. LETICIA can be customized by adding experiment assets like questions, forms, search tasks, and web documents. The next sections will describe how to add these assets.

Questions and Forms

"Questions" are the basic element for assessment in LETICIA. There are five types of questions: `Input`, `Paragraph`, `LikertScale`, `AnonLikert`, and `MultiQuery`. Questions can be loaded into LETICIA using the REST API using JSON format. All questions have three basic elements:

`questionId` (identifier of the question, must be unique), `type` (question type), and `required` (boolean value indicating if the question must be mandatory answered or not).

Questions can be added through a POST request to the `http://localhost:3000/v1/questions` endpoint, while Forms can be added through a POST request to `http://localhost:3000/v1/forms` endpoint. The body of these requests must be in the JSON format described below.

Input

Input questions are answered with single-line string-type responses. These questions have a `title` (the question to perform) and a `hint` (optional text that gives help to get the answer). They can be loaded through LETICIA's API with the following JSON format:

```
{
  "questionId": "question1",
  "type": "input",
  "title": "Who is the author of this blog post?",
  "hint": "Look at the post properties",
  "required": true,
  "answer": null
}
```

Paragraph

Paragraph questions are answered with multi-line string responses, and they share the same properties as the Input questions.

```
{
  "questionId": "question2",
  "type": "paragraph",
  "title": "Why is this page relevant?",
  "hint": "Make a brief explanation",
  "required": false,
  "answer": null
}
```

LikertScale

LikertScale are questions answered in a numeric integer scale. The properties `start` and `stop` set the minimum and maximum of the scale, and `step` sets the numeric difference between the scale points. You can also have text labels at the beginning and end of the scale with `minLabel` and `maxLabel` values respectively. Below each point, the value for it is shown.

```
{
  "questionId": "question3",
  "type": "likert",
  "title": "I know a lot about the topic of the task that I'm going to perform",
  "start": 1,
  "stop": 5,
```

```
  "step": 1,
  "minLabel": "Strongly disagree",
  "maxLabel": "Strongly agree",
  "required": true,
  "answer": null
}
```

AnonLikert

AnonLikert are just like LikertScale questions, but the value for each point is not shown. Useful for scales like the NASA-TLX questionnaire.

```
{
  "questionId": "nasatlx01",
  "type": "anonlikert",
  "title": "Mental demand",
  "hint": "How much mental demand was required by the task?",
  "start": 1,
  "stop": 21,
  "step": 1,
  "minLabel": "Very low",
  "maxLabel": "Very high",
  "required": true,
  "answer": null
}
```

MultiQuery

MultiQuery are questions where the user enters search queries to answer a search task. The text and keystrokes entered in this question format are tracked. The property `queries` set the minimum of queries that must be entered to validate the answer.

```
{
  "questionId": "query1",
  "type": "multiquery",
  "title": "Enter at least 3 queries to obtain information to solve the task",
  "queries": 3,
  "required": true,
  "answer": []
}
```

Forms

Forms are groups of questions. They have two properties: `formId` (string identifier for the questionnaire, required and must be unique) and `questions` (string array of `questionIds` of questions to appear in the questionnaire). Their JSON representation in the LETICIA's API is the following:

```
{
  "formId": "myform",
```

```
  "questions": [ "question1", "question2", "question3" ]
}
```

Loading web documents for a flow

In the context of LETICIA, "documents" are downloaded (and indexed) pages with the web scraper module. Although the web scraper module is intended to work by downloading web pages available on the local network, it can also download pages available on the internet (if available) with limited compatibility. The document parser removes all links, form fields, iframes, and embedded Javascript to ensure that participants don't leave the LETICIA's domain (hence losing track of the actions performed by them). It is the researcher's responsibility to be authorized to use the downloaded web pages in their experiment and to ensure that the web pages are displayed correctly on LETICIA.

Currently, LETICIA has two operations over web documents:

- **POST `/documents/preview`:** The web document is downloaded (but not indexed) and can be accesed through the web browser using the `previewUrl` value on the API response of the request.
- **POST `/documents/index`:** The web document is downloaded and indexed in the inverted index (currently Solr). The document can be accessed via web browser through the `previewUrl` value on the API response of the request or through the search module.

The JSON body structure to preview or index a web document is described below:

```
{
  "docId": "doc01",
  "url": "https://developer.mozilla.org/en-US/docs/Web/API/Element/mousemove_event",
  "locale": "en",
  "title": "Element: mousemove event",
  "snippet": "The mousemove event is fired at an element when a pointing device...",
  "keywords": [
    "javascript",
    "event"
  ],
  "relevant": [
    "task01"
  ]
}
```

The `docId` field is the identifier of the web document (must be unique). The field `url` indicates the URL of the site to download. The `locale` field indicates the document's language on the ISO 639-1 code standard. The `title` field is the document's title is showed on the search results page. The `snippet` field is a fallback value for the search snippet in case LETICIA cannot generate one from the search query. The `keywords` array is a field where some words can be set to enhance the search score for the document from some query terms while performing the search. The `relevant` array indicates the `taskId`s for the tasks where the document is relevant. This JSON structure must be sent as the body of the POST request to LETICIA's API.

Build search tasks

Once questions, forms, and documents are loaded into LETICIA; you can build search tasks to be performed by the participants. Search task builder can be accessed through the `http://localhost:3000/v1/tasks` REST endpoint, where the following actions can be performed:

- **GET `/tasks`:** Get all tasks
- **POST `/tasks`:** Add a new search task
- **GET `/tasks/{taskId}`:** Get one task (by taskId)
- **DELETE `/task/{taskId}`:** Delete one task (by taskId)
- **GET `/task/shuffle`:** Get all tasks shuffled by a Round-Robin assignment to counterbalance any arbitrary order and/or previous setting when tasks were loaded.

More information can be found on the OpenAPI interactive endpoint. The JSON body structure of the request to add a search task is described below:

```
{
  "searchTaskId": "task01",
  "title": "Special flag",
  "description": "Every national flag in the world shares a common geometric
characteristic, except for one country. Which country is it?"
}
```

Where `searchTaskId` is an identifier for the search task (must be unique), `title` and `description` refer to the title of the task and the extended description for the user to perform such task.

Inverted index configuration

To ensure that the Search module is working properly (i.e., stopword and stemming functionalities), you must set up the inverted index locale. This can be done by sending a POST request to the `/search/locale` endpoint. The body of this request should look like this:

```
{
  "locale": "en",
  "addOperation": true
}
```

Where `locale` is the ISO 639-1 language code (like `en` or `es`) for the inverted index locale, and `addOperation` indicates if the configuration must be added or edited to current settings (i.e., set this to `true` if this is the first time that you are configuring the index or `false` otherwise). In case you have loaded web documents before configuring the index, you must reindex all documents at the POST `/documents/reindex` endpoint (with no request body).

Assembling the experimental flow

Once all experiment assets have been loaded (i.e. questions, forms, web documents, and search tasks), you can assemble the experimental flow as a list of stages. An example of an experimental flow in JSON format for LETICIA is shown below:

```
{
  "sessionFlowId": "extended",
```

```
  "flowName": "Extended Challenge",
  "instructions": "In this challenge you will be presented with 3 search tasks, you must
create search queries (in English and using the LETICIA's search engine) to find relevant
documents that can solve these tasks. Once you've been introduced to the task, you must
fill the questionnaires to access the search engine. You have 25 minutes to solve the
challenge.",
  "minDocs": 3,
  "stages": [
    {
      "path": "/consent",
      "timeLimit": -1
    },
    {
      "path":"/demographic",
      "timeLimit": -1
    },
    {
      "path": "/typing-instructions",
      "timeLimit": -1
    },
    {
      "path": "/typing",
      "timeLimit": -1
    },
    {
      "path": "/instructions",
      "params": "extended",
      "timeLimit": -1
    },
    {
      "path": "/extended-challenge",
      "timeLimit": 25
    },
    {
      "path": "/exit-survey",
      "timeLimit": -1
    },
    {
      "path": "/end",
      "timeLimit": -1
    }
  ]
}
```

The following values can be customized in a LETICIA's experimental flow:

- **`sessionFlowId`:** Identifier of the flow (must be unique).
- **`flowName`:** Name of the flow.
- **instructions:** Instructions for the flow when entering a challenge (can be retrieved from the `instructions` stage).

- **`minDocs`:** Minimum amount of documents to bookmark on a `search` stage.
- **`stages`:** An ordered list of stages (expressed as a JSON array). The elements inside this list (JSON objects) must have the following values:
    - **`path`:** The frontend's path to the stage/challenge. Check the "Stages as purposed activities" section for more information on the available frontend paths.
    - **`timeLimit`:** Amount of time (in minutes) required to complete the stage/challenge. If this limit is hit, LETICIA will send the user to the next stage. Set it to `-1` for no time limit.
    - **`params` (optional):** URL parameter (as a string) passed to the stage.

LETICIA's flow builder endpoint can be accessed at `http://localhost:3000/v1/flows`, with the following actions available:

- **GET `/flows`:** Get all flows
- **POST `/flows`:** Add a new experimental flow
- **GET `/flows/{flowId}`:** Get one flow (by flowId)
- **DELETE `/flow/{flowId}`:** Delete one flow (by flowId)

Internationalization

LETICIA's Frontend uses the vue-i18n plugin to translate all message texts to different languages. The initial release has translations for English and Spanish (pull requests on the GitHub repository for other languages are welcome).

The translation template can be found in `client/modules/translations.js`. This file can be modified to add more languages, using the following format:

```
const translations = {
  en: {
    /* English strings */
  },
  es: {
    /* Spanish strings */
  }
};

export { translations };
```

Language identifiers for new languages should follow the ISO 639-1 language code standard (like `fr`, `de`, or `fi`). Language can be changed by setting the LETICIA_LOCALE environment variable when deploying to one of the available languages in the translation template (using the ISO 639-1 language code).

LETICIA's Architecture

The LETICIA platform uses the client-server model to serve content for experimental IIR studies. It is composed of two main components: Frontend and Backend. The frontend is in charge of displaying the search simulation and activities to the client (study participant) on its web browser. At the same time, the backend runs in the server and is in charge of content distribution to clients and capturing their interactions with the frontend (through the REST API). Figure A illustrates the software architecture diagram of LETICIA.

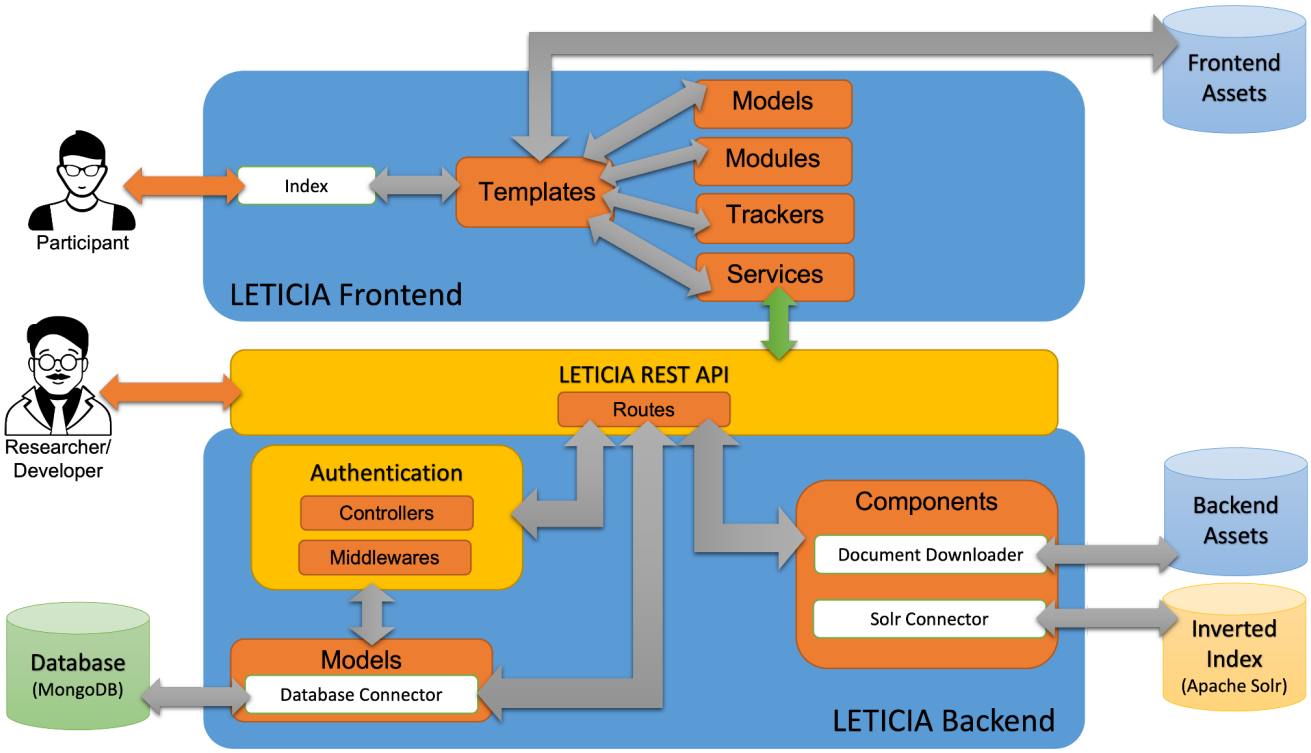


Figure A: LETICIA's architecture diagram

The following sections will describe the chosen architecture and offer a series of developer notes to modify and expand LETICIA.

Frontend

The LETICIA frontend was built on HTML5, CSS3, and ECMAScript JS using Vue.js 2 framework. All frontend builds are minified and polyfilled through Parcel bundler. This frontend implementation is included as a working example of a web application running over LETICIA's REST API. It is possible to use this frontend implementation directly, modify it or implement a custom one according to the experiment needs. All frontend code is stored on `src/client/` directory.

Templates

All views and routes in the frontend are made as Vue.js 2 templates. Currently available templates can be seen in `src/client/templates/` directory.

Frontend Components

The JS Components for the frontend allow the execution of experiment logic directly on the clients' browser. These scripts are distributed on `src/client/services/`, `src/client/modules/`, `src/client/models/`, and `src/client/trackers/` directories.

- **Services** include objects that can be invoked from templates and offer different functionalities to the experiment execution. These services include ActionSender (sends messages with user actions to the backend), Bookmark (in charge of bookmark actions and bookmark list for each user), Timer (in charge of control and display of time on time-constrained stages), Auth (for user authentication), Constants (reads environment variables), and Utils (offers helper functions for other components).
- **Modules** include the implementation of base components that allow the experiment to work. These include the Store (implemented through `vuex`), the Routes (implemented through `vue-router`), the Translations (implemented through `vue-i18n`), the Authentication module, and the EventBus module (that allows message passing between different templates and components during experiment execution).
- **Models** are representations of persisting objects during the experiment. Currently, the only model available on frontend is the User.
- **Trackers** store logic related to user actions performed by the user and registered by the event handlers in the frontend. The trackers are designed as objects containing methods that can be used as event handlers for JS events, returning a message in JSON format that can be sent to the backend. Currently, there are four types of handlers: Action Handlers, Keystroke Handlers, Mouse Handlers, and Scroll Handlers.

Backend

The LETICIA Backend was built using the latest specification of ECMAScript JS, using Node.js 16 and transpiled with Babel. The main dependencies of the backend are `express` (REST API routing and authentication), `mongoose` (connector to MongoDB), `solr-client` (connector to Solr Inverted Index), `website-scraper` (online document downloader), and `cheerio` (document parsing). These dependencies are available on the NPM repository. All backend code is available on `src/server/` directory. The main source code file for the backend can be found on `src/server/index.js`. The backend comprises four modules: REST API, Models, Authentication, and Components.

REST API

All interactions between the backend and other components are mediated by the REST API. This module has been documented through the OpenAPI standard, allowing direct interactions between the researcher/developer and the backend through a web interface. By default, you can access the OpenAPI documentation through your web browser on `http://localhost:3001/openapi/` where you can interact with the REST API and see examples and input schema of the different endpoints available. Figure B shows an example of the “Preview document” endpoint.

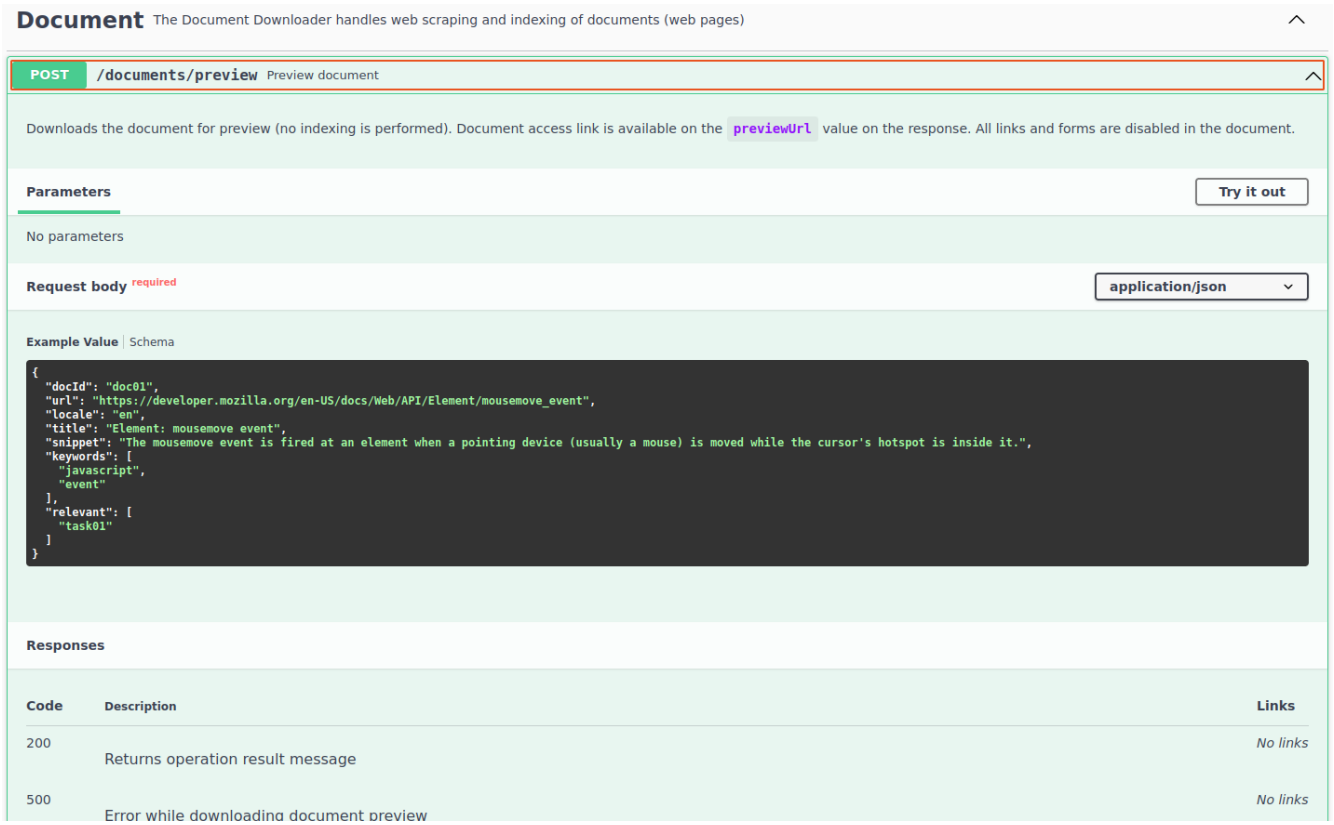


Figure B: Performing a POST query (downloading a document for preview purposes) through the OpenAPI documentation platform

To interact with the API through the OpenAPI interactive documentation, follow these steps:

1. Enter the documentation interface through the browser

2. From the “Servers” dropdown box, select the LETICIA server endpoint where the requests are performed (by default, this is set to `http://localhost:3000/v1`)
3. Select a request to perform to the API and check the parameters, request body example, and schema
4. To perform the request, click the “Try it out button” and customize the parameters and request body (all requests must be made in JSON format)
5. Press “Execute” to perform the request, and the result should be returned right away

In case you need to expand the API, follow the examples available in `src/server/routes/` to create new endpoints. The file `src/server/openapi.js` contains the definition of the OpenAPI specification, implemented through the `swagger-ui-express` and `swagger-jsdoc` NPM packages.

Models

LETICIA’s Models module is in charge of the connection to the MongoDB database and the load/save of database entries through the Mongoose Object-Relational Mapper. All model files are available on `src/server/models/` directory and `src/server/db.js` stores the definition of databases.

LETICIA uses two different databases on the same connection to store its data. By default, they are called `leticia-data` and `leticia-user`. The former stores all experimental data (like session flows, distributed content, and participants’ interactions with the platform), while the latter stores all identifiable credentials for the participant (like his/her email and Single Sign-on provider). This design allows researchers to use the experimental database for analysis without compromising the anonymity of the participants.

You can customize the connection to the MongoDB instance during deployment by modifying the `MONGODB_HOST`, `MONGODB_PORT`, `MONGODB_DATA_DB`, and `MONGODB_USER_DB` environment variables.

Authentication

LETICIA offers two authentication methods: Email and Single sign-on (SSO). LETICIA’s authentication code is available in `src/server/controllers/` and `src/server/middlewares/` directories, and the `src/server/oauth.js` file. You can enable/disable these authentication methods by setting the following environment variables on deployment: `ENABLE_EMAIL_LOGIN`, `ENABLE_GOOGLE_LOGIN`, and `ENABLE_FACEBOOK_LOGIN`.

SSO login is currently implemented for Gmail and Facebook accounts. For these methods to work, you need to access the Google Cloud Platform Console or the Facebook Developers Console and build a web application that can manipulate OAuth credentials. When this is set, you get a Client ID and a Secret ID that can be used to configure LETICIA’s SSO logins through the following environment variables: `GOOGLE_CLIENT_ID`, `GOOGLE_CLIENT_SECRET`, `FACEBOOK_CLIENT_ID`, and `FACEBOOK_CLIENT_SECRET`.

By default, redirection URLs for SSO login are `http://localhost:3000/v1/auth/googleRedirect` and `http://localhost:3000/v1/auth/facebookRedirect` respectively.

Backend Components

LETICIA's backend Components module manipulates all the logic required for the search engine simulation. To this date, it stores the Document Downloader and Solr Connector submodules. The code of this module is available in `src/server/components/` directory.

The Document Downloader is responsible for scraping and cleaning websites indexed by Solr and displayed by LETICIA's search engine. Please bear in mind that to avoid CORS issues by the client's browser while displaying documents in the search engine, all links and embedded JS code are removed from the documents at download time.

Solr Connector is in charge of indexing and retrieval of document results through the Solr Inverted Index. You can customize the connection to the Solr instance during deployment with the `SOLR_HOST`, `SOLR_PORT`, and `SOLR_CORE` environment variables.

Modifying LETICIA's Source Code

Languages and Frameworks

Regarding the programming languages used in LETICIA, most of the code base uses JavaScript following the ECMAScript 6 standard (ES6). For the frontend, Vue templating format is used for the views (composed by HTML5, CSS3, and JavaScript ES6).

LETICIA uses the following major frameworks:

- Backend:
  - Node.js v16.x (main backend framework
  - Express.js v4.x (API routing)
- Frontend:
  - Vue.js v2.x (main frontend framework)
  - Vuex v3.x (user data storage in client browser)
  - Vue Router v3.x (view routing)
  - Vue I18n v8.x (internationalization support)
  - Twitter Bootstrap v4.x with BootstrapVue (visual style and layout)
  - Lodash v4.x (utilities)
- Connectors to external dependencies:
  - Mongoose (connector to MongoDB)
  - Solr client for Node.js (connector to Apache Solr)

Directory description

LETICIA has the following directory layout:

```
leticia
├── docs                  // LETICIA Documentation
├── extras                // Extra scripts
│   └── databaseScripts    // Database cleaning and populating scripts
└── src
      ├── client          // Frontend code
      │   ├── assets       // Frontend file assets
      │   ├── models       // Database model representations from backend
      │   ├── modules      // Frontend core modules (auth, event bus, routes, store, translations)
      │   ├── services     // Frontend core services (mainly connectors with API)
      │   ├── templates    // Frontend Vue templates
      │   │   ├── auth           // Authentication templates
      │   │   ├── formElements   // Form element templates
      │   │   ├── hubs           // User hub and admin hub templates
      │   │   ├── queryPlanning  // 'Short challenge' templates
      │   │   ├── questionnaires // Form templates
      │   │   └── search         // 'Extended challenge' templates
      │   └── trackers      // Frontend tracking services (keyboard, mouse, and scroll)
      └── server          // Backend code
          ├── assets       // Backend file assets
```

```
│   ├── documents // Downloaded documents (scraped web sites)
│   └── preview   // Preview documents
├── components    // Backend main components (web scraper and inverted index connector)
├── config        // Authentication configs
├── controllers   // Authentication controllers
├── middlewares   // Authentication middlewares
├── models        // Database model files (Mongoose)
└── routes        // API routes (Express.js)
```

Considerations

LETICIA is released as open-source software and can be modified to tailor researchers' needs. In case of need to edit LETICIA's source code, please fork this repository. Pull requests for new functionalities and translations are welcome.

To deploy LETICIA locally in development mode, run the following command on the project's `src/` directory:

```
npm run dev
```

Frontend

All frontend files are located on the `src/client/` directory. They can be either .js or .vue files. Assets for images, JSON files, and PDF documents are stored on `src/client/assets/` directory. Translation strings are stored on `src/client/modules/translations.js` file. The main frontend file is located on `src/client/index.html`.

Creating or editing view templates

All view templates must follow the Vue.js v2 format and must be located in the `src/client/templates/` directory. View templates can import modules, services, or trackers if needed. For visual styling and layouts, the BootstrapVue framework must be used.

This is an example of a view template:

```
<template>
  <!-- HTML code -->
  <b-container id="new-template">
    <b-row class="text-center">
      <b-col>
        <h1 class="main-title">
          My new template
        </h1>
        <img :src="newImage"></img>
      </b-col>
    </b-row>
  </b-container>
</template>
```

```
<script>
/* JS code */
import Image01 from '../assets/image01.jpg';

export default {
  name: 'new-template',

  /* Reactive variables */
  data() {
    return {
      newImage: Image01
    }
  },

  /* Computed values */
  computed: {
    // Checks if user is logged in (or not)
    loggedIn() {
      return this.$store.state.auth.status.loggedIn;
    },
    // Returns current user object
    currentUser() {
      return this.$store.state.auth.user;
    }
  },

  /* Code that runs when the view template is created */
  created() {
    // (...)
  },

  /* Code that runs before the template is rendered in browser */
  mounted() {
     // (...)
  },

  /* View auxiliary functions */
  methods: {
    isUserLogged(): {
      return this.loggedIn;
    }
  }
}
</script>

<style scoped>
/* CSS code */
.main-title {
  font-weight: bold;
```

```
  color: #FF0000;
}
</style>
```

Event Bus

LETICIA uses the Event Bus pattern to transfer data between view components, especially between the main activity window and the navbar component. The Event Bus pattern is very similar to the publish-subscriber pattern, using a string identifier to determine to which event the event handler must react.

This is an example of an event emitter (using the `leticia-current-task` identifier):

```
import EventBus from '../modules/eventBus';

EventBus.$emit('leticia-current-task', { currentTask: true });
```

This is an example of an event handler (using the `leticia-current-task` identifier):

```
import EventBus from '../modules/eventBus';

EventBus.$on('leticia-current-task', (data) => {
  this.showCurrentTask = data.currentTask;
});
```

Backend

All backend code is located on the `src/server/` directory, using JavaScript with ES6 syntax.

Extending the REST API

All endpoints of LETICIA's REST API are located in the `src/server/routes/` directory. Endpoints are implemented using Express.js and use the async/await directive to run asynchronous code. The `src/server/api1.js` imports all endpoints and makes them available to the server/index.js main file.

Extending the database

In case of need to add additional collections/tables to the database, use the Mongoose framework and follow the schema available on `src/server/models/` and `src/server/db.js`. LETICIA by default generates two databases: The "user" database for all login and sensible data about the participants and the "data" database for all the experimental data obtained from participants' usage of the platform.

OAuth login

LETICIA is compatible with OAuth SSO Login using one of these two providers: Google or Facebook. All relevant code can be found in the following files/directories:

- src/server/config/
- src/server/controllers/
- src/server/middlewares/
- src/server/oauth.js

Important considerations

This section details essential aspects to be considered while deploying and/or developing code for LETICIA.

Deployment

- LETICIA's backend is in charge of deploying the frontend and OpenAPI interactive documentation. These modules can be disabled on deployment by setting `ENABLE_FRONTEND` or `ENABLE_API_DOCS` environment variables to `false`.
- By default, LETICIA uses the following ports on deploy:
    - 3000 (Frontend and REST API)
    - 3001 (OpenAPI documentation)
    - 8983 (Solr Index on Native mode) or 3011 (Solr Index on Docker mode)
    - 27017 (MongoDB on Native mode) or 3010 (MongoDB on Docker mode)
- It is highly recommended to block remote access to both Solr and MongoDB ports by using a firewall. You can block those ports with `ufw` in Native mode or `IPTABLES` on Docker mode. In both cases, an external firewall from the VPS provider can also work.
- You can use a reverse proxy to redirect the Frontend and REST API port to something else (like 80 or 443). Check here for an example template for a reverse proxy using Nginx.
- All frontend assets are stored on `src/client/assets/` folder. In there you may change all images, JSON files and PDF documents with custom ones.
- In case of deploying LETICIA through Docker, it's highly recommended to load all custom assets in Dev mode before building the Docker image, to ensure they are copied and imported into the resulting image.
- All backend assets are stored on `src/server/assets/` folder. In there you may find the previewed web document and the downloaded ones, in case of need to backup them.

Development

- Due to Cross-Origin Resource Sharing (CORS) policies on all evergreen web browsers, in order for frontend trackers to work inside iframes containing web documents from the backend, both frontend and backend must run on the same URL protocol, address, and port (by default `http://localhost:3000`). If you do not need to track actions inside the web documents, you can use a different port or domain for backend and frontend.
- Related to the previous point, by default, the document downloader of LETICIA removes all links, iframes, and embedded JS logic from scraped sites. This is done to avoid users from escaping the LETICIA's URL domain and therefore disabling all frontend trackers due to CORS policies on the web browser. The document cleaning behavior can be adjusted on the `cleanDocument()` method on `src/server/components/documentDownloader.js` file.
- Login is handled through JWT (JSON Web Token) and the Passport.js package. All session status is stored on the browser LocalStorage with the vuex v3 package.
- The interaction capture on the frontend is handled directly from the browser by using Javascript client events. Captured events are sent from the Action Sender service (`src/client/services/ActionSender.js`), while custom trackers are in charge of generating the JSON objects sent as body request to the API (these trackers are located in

`src/client/trackers/` directory). The captured interactions are events triggered directly by the Vue templates, who call the trackers and the Action Sender.

- In case of needing to store custom events not directly covered by the API, LETICIA has the BrowserEvents endpoint to store any kind of Javascript client event on the browser.
    - **POST `/events`:** Store a single browser event (from JSON object as body request).
    - **POST `/events/buffer`:** Store multiple browser events (from a JSON array as body request).

References

Hart, S. G., & Staveland, L. E. (1988). Development of NASA-TLX (Task Load Index): Results of empirical and theoretical research. In *Advances in psychology* (Vol. 52, pp. 139-183). North-Holland.

Byers, J. C., Bittner, A., & Hill, S. (1989). Traditional and raw task load index (TLX) correlations: are paired comparisons necessary? *Advances in Industrial Ergonomics and Safety I*, *1*, 481–485.

Stroop, J. R. (1935). Studies of interference in serial verbal reactions. *Journal of experimental psychology*, *18*(6), 643.

Gwizdka, J. (2008). Cognitive load on Web search tasks.

Documentation Changelog

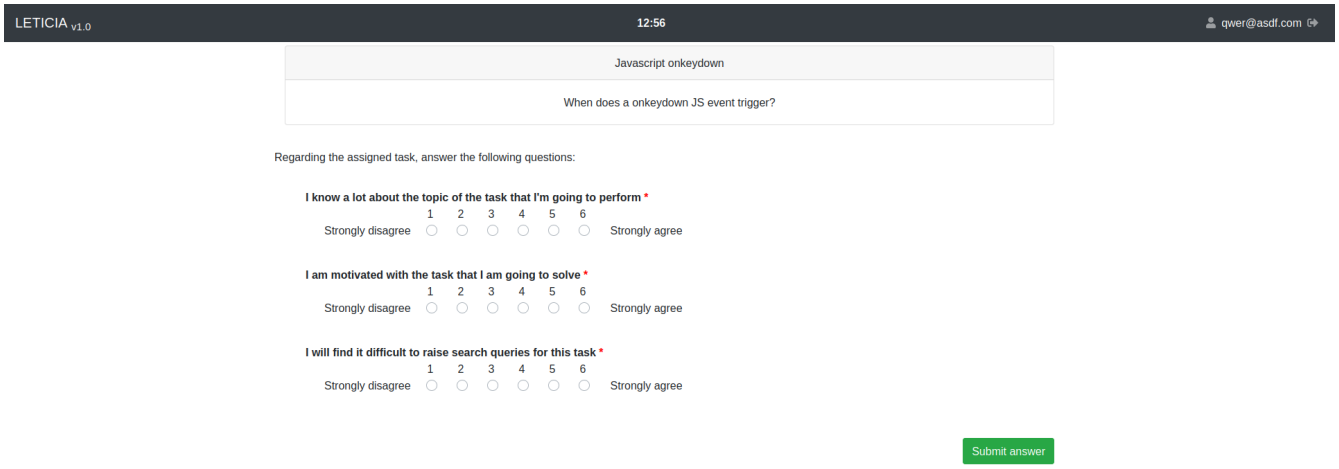
2022-06-20 (v1.0.0): First draft of documentation
2022-09-25 (v1.0.1): First complete version of documentation
2022-10-06 (v1.0.2): Amendments for SoftwareX publication

Appendix

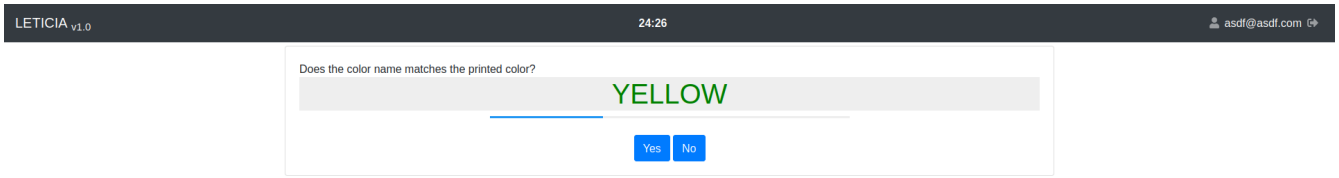
Screenshots

The following section showcases some screenshots of the LETICIA platform on the frontend side:

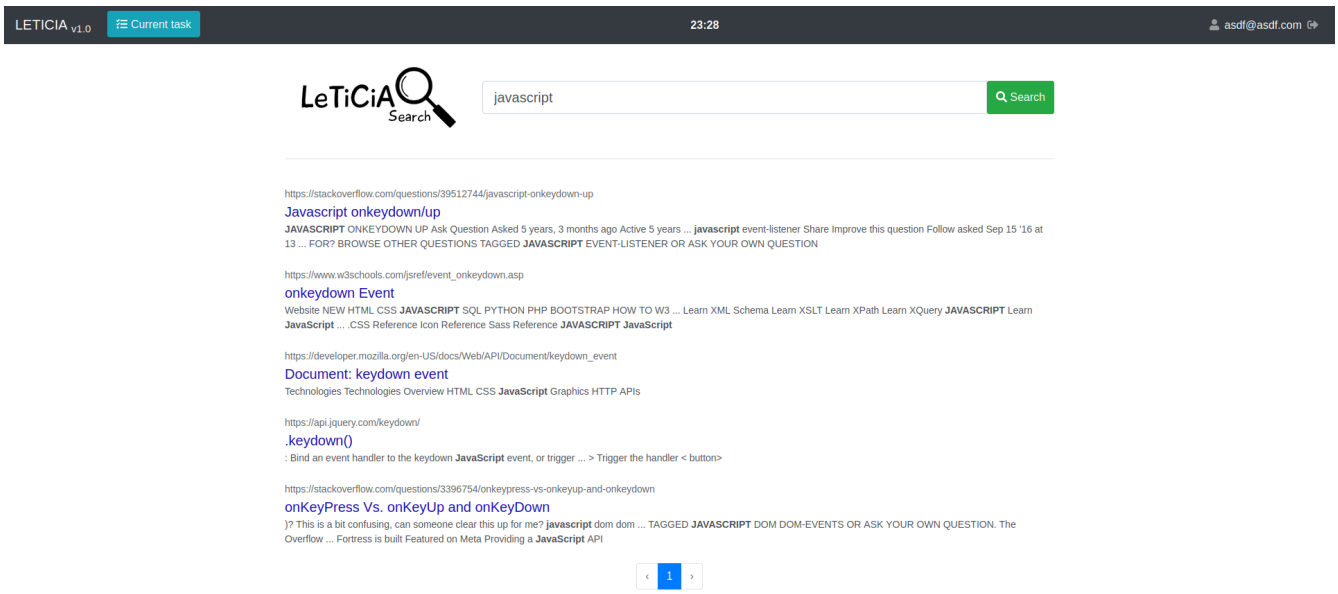
Forms



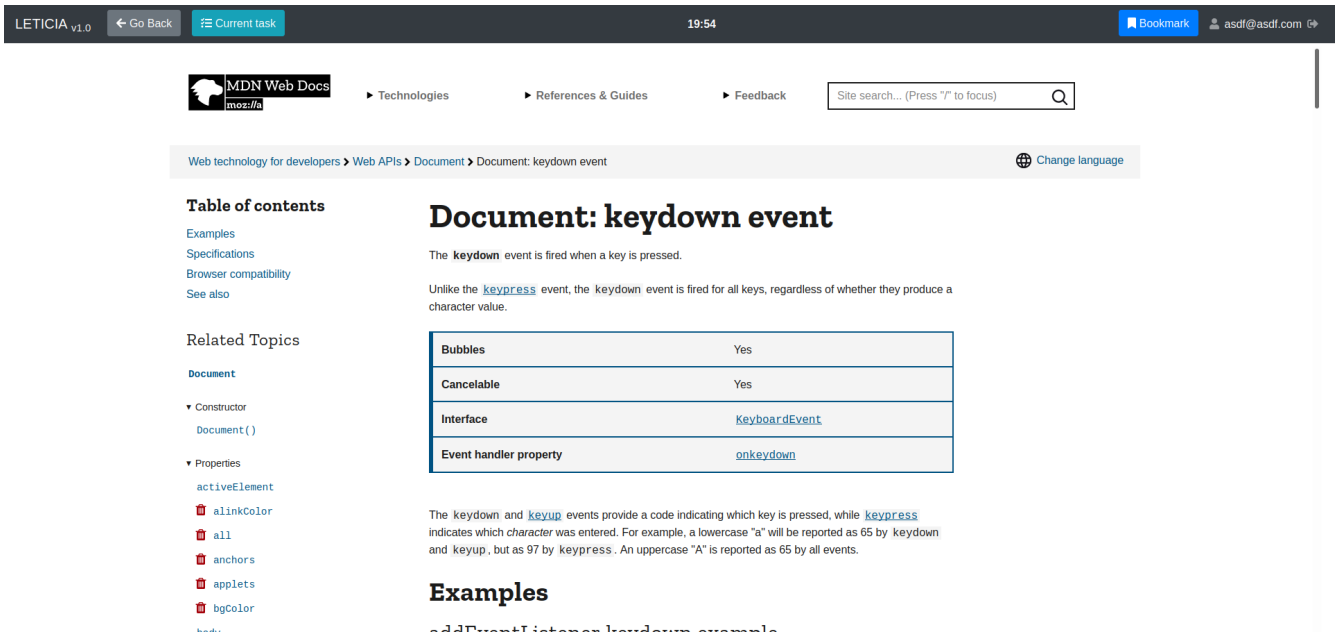
Stroop Test



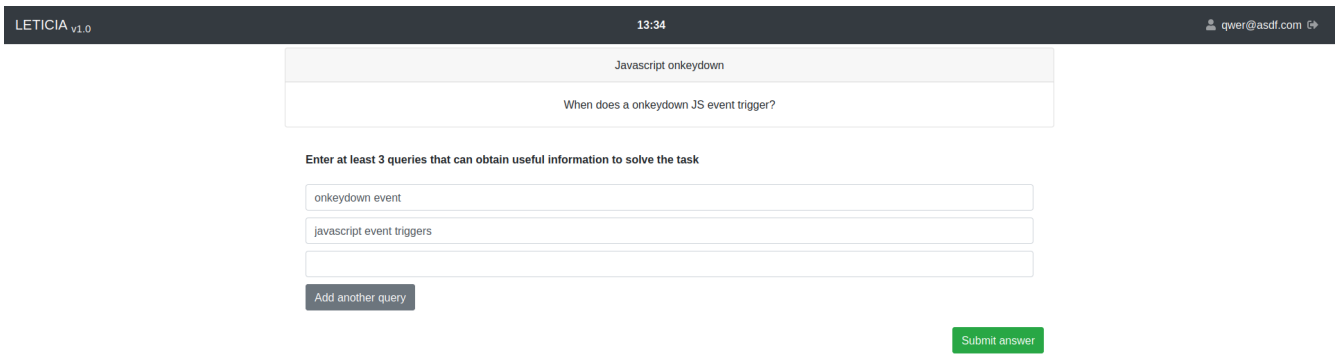
Search Engine Results Page (Search stage)



Document Viewer (Search stage)



Query Planning



Typing Test