Final Exercise Specification: Survey Server with AI Summarization

Concept

Build a Survey Server that allows users to share opinions on various areas in free text, which is then analyzed and summarized by AI.

- A survey is defined by a **survey guidelines** - basically a prompt to a LLM explaining what the survey is about, which response domains are permitted, what type of responses are permitted, and how to summarize the survey.
- For example:
    - **Survey area**: improving the elementary school playground
    - Survey expiry date: 20 days
    - **survey question**: how would you like to improve the school playground?
    - **permitted domains**: what we want to keep, what we want to improve
    - **permitted responses**: use proper language, limit your response to two paragraphs.
    - **summary instructions**: make the summary readable by 6-8 graders, and introduce some humor.
- Another example:
    - **Survey area**: SWOT (strength/weakness/opportunity/threat) analysis of our organization
    - Survey expiry date: 20 days
    - **survey question**: provide a SWOT analysis of our organization from your viewpoint
    - **permitted domains**: strengths, weaknesses, opportunities, threats
    - **permitted responses**: no need to relate to competitor Z who went out of business.
    - **summary instructions**: summarize the insights without adding any interpretation on the position of other competitors.
- The server stores all information about surveys and responses in a database.
- Users are allowed to:
    - create surveys in any area
    - contribute to any survey running on the server, and
    - change their responses on open surveys until surveys expire or are closed (only by the creator).
- The server has a natural language search endpoint, which operates as follow:
    - E.g., find me all the surveys that relate to food.
    - In the background, the server will create a context (all surveys with ids), prepare a prompt to LLM to search the surveys, and send the request to LLM.
    - The result would be a list of survey ids, and a short reason why this survey matches the query.

```
const systemPrompt = `
```

```
You are an AI assistant tasked with searching ALL surveys matching a natural
language query.

You will receive a JSON object with the following fields:

- query: the natural language search query
- surveys: an array of survey objects, each containing all survey fields (e.g.,
_id, title, area, question, guidelines, permittedDomains, permittedResponses,
summaryInstructions, expiryDate, responses, etc.)

Your task is to return a JSON array of objects, including ALL surveys matching
the query, where each object has:

- id: the survey ID (as string)
- reason: a short explanation of why this survey matches the query

Return only the JSON array of these objects.
`;
```

- Server creator can operate on Summarize survey endpoint:
    - Works in the same manner
    - The summary is stored with the survey, and is then viewable by all users.
    - The creator can ask for a summary at any point in time, and show or hide it at will.
- Validate user responses - upon request from the survey creator, based on the instructions in the prompt guidelines and the proposed response, asking the LLM to verify adherence.
    - The result would be a list of `{ surveyId, responseId, reason }`
    - Survey creator can remove those responses if required.

Actor	**Behavior**	**Conditions/Notes**
Any authenticated user	Create a new survey	Defines guidelines at creation.
Survey creator	Manually close a survey / change expiry time	Only the creator can close before expiry.
Any authenticated user	Submit a response to an open survey	
Responding user	Update/remove their own response	Only while the survey is open.

Server	Enforce survey expiry	No new or updated responses accepted after expiry time.
Survey creator	Check survey responses meet guidelines	Use LLM validation; returns a list of URIs to surveys and a short paragraph that explains violation.
Survey creator	Remove any response at will	E.g., when responses violate guidelines
Survey creator	Trigger AI summarization	Can request at any time. Summary generated based on stored responses
Survey creator	Control visibility of summary	Summaries hidden by default; creator may show or hide.
Any authenticated user	Search surveys via natural language query	Returns URIs to matching surveys.

Authentication flow

Phase	Step	Endpoint	What Happens	Important
Registration	1	`POST /auth/register`	User sends `{ username, email, password, registrationCode }`	`registrationCode` is **mandatory**
	2		Server **validates** fields with Joi, including `registrationCode`	Schema includes registrationCode field
	3		Server **checks** if `registrationCode ===`	If wrong → reject 403 Forbidden

			`process.env.REGISTRATION_SECRET`	
	4		If correct, server **hashes password** (bcrypt) and **creates User** `{ username, email, passwordHash }`	Password never stored in plaintext
	5		Server responds: `{ message: 'Registration successful' }`	**No JWT yet**
Login	6	`POST /auth/login`	User sends `{ email, password }`	No `registrationCode` needed anymore
	7		Server finds user by email, compares hashed password	If wrong → 401 Unauthorized
	8		If password OK, server **issues JWT** `{ userId, username }`	Signed with `process.env.JWT_SECRET`
	9		Server responds: `{ token: <jwt> }`	Client stores the token
Authenticated Use	10	Protected APIs (e.g., create survey)	Client sends token in `Authorization: Bearer <token>` header	Server verifies token in middleware

Prompts

Store all prompts in a designated directory and load them /validate presence when server loads.

```
project-root/
```

```
├── prompts/
│   ├── searchPrompt.txt
│   ├── validatePrompt.txt
│   └── summaryPrompt.txt
```

Tech Stack Requirements

- Node.js, Express
- Swagger documentation
- MongoDB with Mongoose ODM
- JWT authentication
- CORS configuration
- Joi for input validation
- Winston for structured logging
- Morgan for HTTP access logs
- Jest for test runner
- Chai for assertions
- Supertest for API testing
- Faker for generating fake data
- Mocks for external services
- dotenv for environment variable management (only dev, test required)

Security Requirements

- See registration and authentication spec
- API keys (e.g., OpenAI) must be securely managed through .env files.
- No hardcoded keys in the codebase.

Expectations

- Modular code structure: Controllers, Services, Models.
- Fully RESTful API design:
    - Proper HTTP verbs and status codes.

Logging & validation

- Use `winston`
- `{ error: { code, message } }` in all 400/403/500 responses.
- Failed LLM /DB - internal server error
- Joi used for schema enforcement on all input bodies.

- Validation errors must return 400 with clear error messages.
- All schemas must be centrally defined for reusability.

Testing

- Clear separation of testing and dev environments [no need for more env]
- use `mongodb-memory-server` for all tests
- Use environment flags, e.g, USE_MOCK_LLM=true in .env.test.
- Test coverage:
    - Every major function (creating a survey, submitting an opinion, summarizing, etc.) must be tested
    - Every API endpoint must be tested with Supertest
    - Unit tests for business logic.
    - Test Independence: Each test sets up and tears down its own data, so tests don't interfere with each other.
    - No Side Effects: Changing one test won't break another.
    - Set up also coverage testing using jest, and aim for 70% coverage
- **Mocking** of LLM calls during tests
    - Replace LLM call with a **simple, fake static function** that returns a **random or canned summary**.
- Never call real external APIs in tests.
- Limit number of stored opinions per survey + make sure that surveys are short not to overload LLM……
- You can use any free LLM (see www.openrouter.ai, choose price = 0), or a key that we will provide for teams.

Required Test Artifacts:

- All test files must be named: `*.test.js`
- Place them either:
    - next to the files they test (same folder), or
    - in a `tests/` folder with a clear structure
- Use a `__mocks__/` folder to hold:
    - a mocked `llmService.js` that returns static results
    - any other fake services if needed

GUI - Bonus

- You may receive up to 15% bonus for putting together a GUI of your choice - over any framework you like, but the code must be readable and as much as possible follow MVC pattern.