INTRODUCTION TO

LangChain.Js

Luigi Brandolini

SYLLABUS

LANGCHAIN.JS

- Introduction
- Core features
- How to use in a TypeScript project
- Integrating LangSmith
- A concrete scenario of usage
- Final considerations

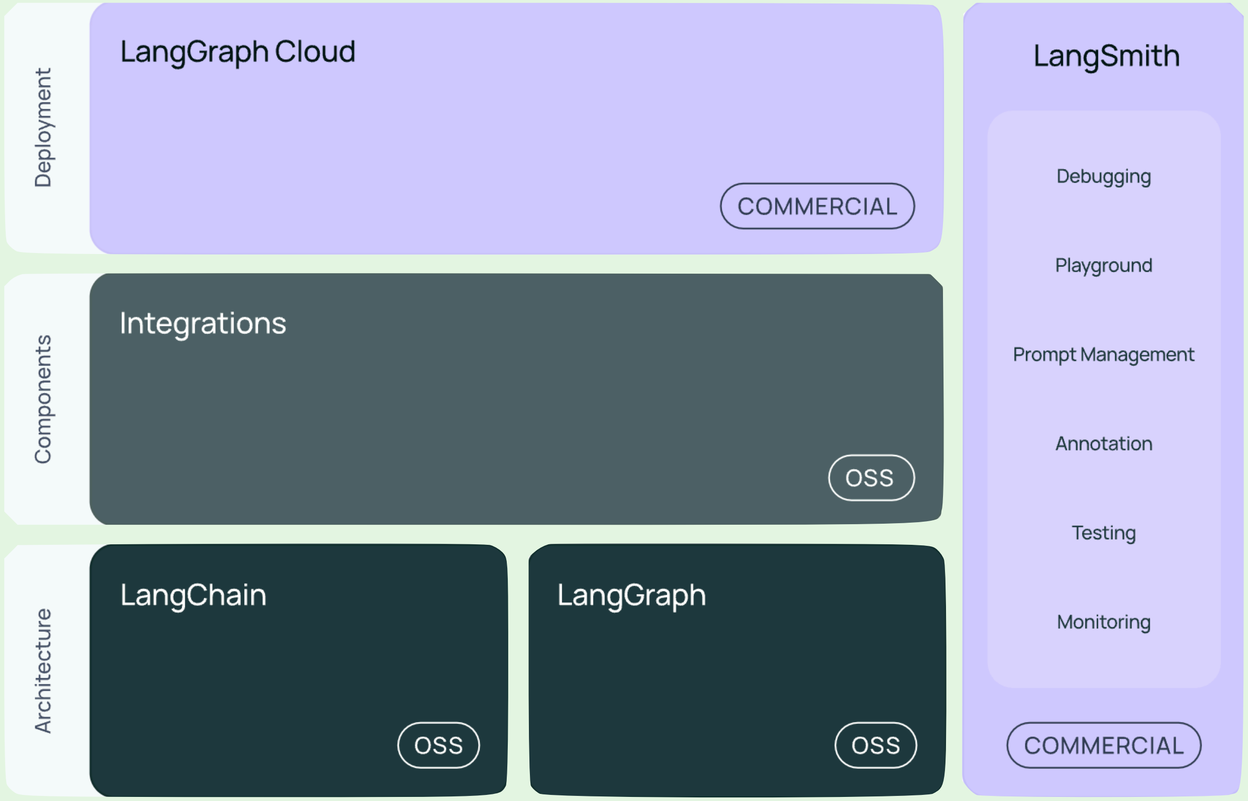
INTRODUCTION

LANGCHAIN.JS

- A JavaScript/TypeScript framework for building applications powered by Large Language Models (LLMs).
- Focuses on prompt templating, managing memory, and interacting with external APIs.
- Suitable for building chatbots, document analysis tools, and generative apps.

INTRODUCTION

LANGCHAIN.JS ECOSYSTEM



INTRODUCTION

LANGCHAIN.JS ARCHITECTURE

- The framework consists of the following open-source libraries:
  - `@langchain/core`: base abstractions and LangChain Expression Language.
  - `@langchain/community`: third party integrations (e.g. `@langchain/openai`, `@langchain/google-genai`, `@langchain/anthropic`, etc.)
  - `langchain`: chains, agents, and retrieval strategies that make up an application's cognitive architecture.
  - LangSmith: A developer platform that lets you debug, test, evaluate, and monitor LLM applications.
  - LangGraph.js: Build robust and stateful multi-actor applications with LLMs by modeling steps as edges and nodes in a graph.

CORE FEATURES

- **Chains: Combine multiple LLM calls into structured workflows by means of LCEL syntax.**
- **Memory: Maintain conversational context across interactions.**
- **Tooling: Integrate with APIs, databases, or custom tools.**
- **LLM Integration: Works with major LLMs like OpenAI GPT, Google Gemini, etc.**

INTEGRATING LLM

- Google Gemini API: connect using API keys and endpoints.
- LangChain abstracts prompt engineering for Gemini models.
- Customizable prompts, memory, and chaining tailored for Gemini's capabilities.

USING LANGCHAIN.JS WITH TYPESCRIPT

- **Benefits:**
  - **Type safety for robust and maintainable code.**
  - **Easier debugging with static analysis.**
  - **Seamless integration with modern TypeScript tooling.**

SETTING UP A NEW PROJECT

- **Pre-requirement: the latest Node.Js version must be installed on the OS.**
- **Create a new Node.js project with the `package.json` descriptor:**

```
mkdir langchain-ts-app

cd langchain-ts-app

npm init -y
```

SETTING UP A NEW PROJECT

- **Install the required dependencies:**

```
npm install langchain @langchain/core @langchain/google-genai dotenv
```

- **Install the required TypeScript and Node.js types:**

```
npm install -D typescript @types/node ts-node
```

SETTING UP A NEW PROJECT

- **Set up TypeScript with the `tsconfig.json`:**

```
npx tsc --init
```

SETTING UP A NEW PROJECT

- **Configure properly the `tsconfig.json`:**

```
{
  "compilerOptions": {
    "target": "ES2017",
    "module": "CommonJS",
    "strict": true,
    "esModuleInterop": true,
    "outDir": "./dist"
  },
  "include": ["src/**/*"],
  "exclude": ["node_modules"]
}
```

SETTING UP A NEW PROJECT

▶ Create the environmental configuration file '`.env`'

▶ It will contains some essential properties, like the Google Gemini's API Key and the GEMINI MODEL that we want to use:

```
GEMINI_API_KEY=<GOOGLE_GEMINI_API_KEY>
GEMINI_MODEL=gemini-2.0-flash-exp
```

CREATE THE APPLICATION STRUCTURE

- **The directory structure could be like this:**

```
langchain-ts-app/
├── src/
    ├── main.ts
    ├── langchain.ts
├── .env
├── package.json
├── tsconfig.json
```

INITIALIZE THE APPLICATION COMPONENTS

▶ langchain.ts:

```
import * as dotenv from "dotenv";
import { ChatGoogleGenerativeAI } from "@langchain/google-genai";
import { PromptTemplate } from "@langchain/core/prompts";

dotenv.config();

const GEMINI_API_KEY = process.env.GEMINI_API_KEY;

if (!GEMINI_API_KEY) {
    throw new Error("Missing Google Gemini API Key");
}

const GEMINI_MODEL = process.env.GEMINI_MODEL;

if (!GEMINI_MODEL) {
    throw new Error("Missing Google Gemini Model");
}

// Create an Google Gemini LLM client instance
const llm = new ChatGoogleGenerativeAI({
    model: GEMINI_MODEL,
    maxOutputTokens: 2048,
    apiKey: GEMINI_API_KEY,
});

// Define a prompt template
const promptTemplate
 = new PromptTemplate({
    inputVariables: ["question"],
    template: "Answer the following question concisely: {question}",
});

// Define the LLM fetch request function
export const fetchRequest = async (text: string) => {

    // Replace the 'question' placeholder with the actual question
    const formattedPrompt = await promptTemplate.format({
        question: text,
    });

    // Execute the prompt
    return await llm.invoke(
        formattedPrompt
    );
};
```

INITIALIZE THE APPLICATION COMPONENTS

▶ **main.ts:**

```ts
import { fetchRequest } from "./langchain";

const main = async () => {
  const question = "Tell me a joke.";
  const response = await fetchRequest(question);
  console.log("Response:", response.content);
};

main().catch((err) => {
  console.error("Error:", err);
});
```

APPLICATION RUNNING

- **Dev running:**

```
npx ts-node src/main.ts
```

- **Production running:**

```
npx tsc
node dist/main
```

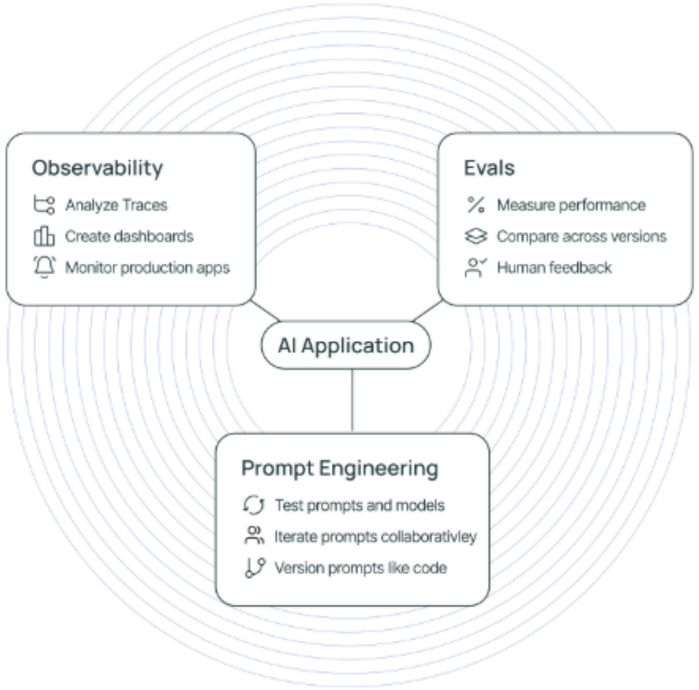
LANGSMITH

- LangSmith is a platform for building production-grade LLM applications:

`https://smith.langchain.com/`

- It allows to closely monitoring and evaluating our applications.
- It is effectively supported by LangChain.Js.

LANGSMITH



Observability

Analyze traces in LangSmith and configure metrics, dashboards, alerts based on these.

Evals

Evaluate your application over production traffic — score application performance and get human feedback on your data.

Prompt Engineering

Iterate on prompts, with automatic version control and collaboration features.

LANGSMITH INTEGRATION

- Generate a LANGSMITH API KEY from the official web-site:

`https://smith.langchain.com/`

- Add the API KEY previously created into `'.env'` configuration file:

```
GEMINI_API_KEY=<GOOGLE_GEMINI_API_KEY>
GEMINI_MODEL=gemini-2.0-flash-exp
LANGSMITH_API_KEY=<LANGSMITH_API_KEY>
```

LANGSMITH INTEGRATION

▶ Define a tracer object inside `langchain.ts` and use it in the invoke function:

```
import { LangChainTracer } from "@langchain/core/tracers/tracer_langchain";

...

const tracer = new LangChainTracer({
  projectName: "langchain-ts-app"
});

...

return await llm.invoke(
    formattedPrompt,
    { callbacks: [tracer] }
);
```

LANGSMITH MONITORING

▶ **In the official LangSmith's web-site, is possibile to check application's logs in the tab "Tracing Projects", inside the section "Observability".**

▶ **Each project created contains the *traced runs* which can be analysed along with statistical measurements.**

▶ **Each run represents a single project execution.**

FINAL CONSIDERATIONS

- **Pros of using LangChain.js:**
  - **Simplified interactions with LLMs, clear templating mechanisms, workflows, and modularity.**
  - **High reliability thanks to the strong typing system offered by TypeScript which ensures error-free and maintainable code.**
  - **Insightful monitoring with LangSmith integration which provides tools for monitoring, logging, and observability, essential for refining AI systems.**

- **Cons of using LangChain.js:**
  - **Limited documentation**
  - **Less mature ecosystem, compared to LangChain for Python**
  - **More performance overheads, compared to LangChain for Python (due of the nature of JavaScript)**
  - **Rapidly Changing API**

CREDITS

Luigi Brandolini

PhD Candidate | Senior Software Engineer

`luigi.brandolini@imt-atlantique.fr`