

Contents

Editorial	2
Architecture: API REST & HTTP	3
Architecture REST	3
Utilisation HTTP	4
Convention de création pour les API	5
Données manipulées par les services	6
Gestion des erreurs	8
Stockage des erreurs (logs)	8
Version des API	9
Documentation API	10
References	11
Backend For Frontend : Notes	12
Avantages	12
Inconvénients	12
Implémentation	12
Référence	13
Notes on Large Language Models	14
Common applications of LLM	14
What are the types of language modelling?	14
What is PLM?	15
References	17
Mentions Légales	18

Editorial

Le-Point-Technique est un journal annuel regroupant les travaux (principalement étudiants ou en tout cas a visée pédagogique), Ces travaux sont généralement sous la forme de tutoriels ou de notices explicatives.

Ce premier numéro contient trois travaux. Un article sur les bonnes pratiques à respecter avec les API REST. Une notice explicative sur le pattern d'architecture Backend For Frontend. Et pour finir, une notice explicative sur les Large Language Models (LLM).

Bonne lecture !

Grégoire Cattan

France, 2023

Architecture: API REST & HTTP

Evan, David

Le-Point-Technique, November/2022

abstract: Le présent article vise à présenter les conventions communes à la construction des API REST. Elle est issue d'un ensemble de bonnes pratiques communément appliquées et d'expériences accumulées sur la création des API REST.

keywords: API, REST, HTTP

Architecture REST

Manipulation des ressources

Lors de la conception des API, les règles suivantes s'appliquent :

- Les URL doivent être construites conformément aux règles & bonnes pratiques de l'architecture REST. Les identifiants des ressources doivent être passés en route param.
- Les ressources présentes dans les URLs seront systématiquement écrites au pluriel, même si une seule ressource est accessible. *Exemple :* `/contract/v1/contracts/123456789`
- Toutes les ressources d'un même service doivent impérativement partager un vocabulaire commun. Un champ représentant une donnée (exemple : Prix HT) doit disposer de la même "traduction", peu importe l'API utilisé au sein de ce service et / ou le modèle utilisé, tant que la donnée possède le même sens.

Convention de nommage des URLs

Les conventions de nommages s'appliquent principalement à la nomenclature des URLs accessibles et composant les services API.

- Utilisation de la convention de nommage Kebab Case.
- Utilisation de la langue **anglaise** pour le nommage des services, fonctions, attributs, ressources ...
- Des **ressources** (et non des fonctions) doivent être utilisés dans les URLs (exemple `/contracts` et non `/getallcontracts`)
- Le nom des attributs composant une ressource devrait être différent des noms des champs de la base de données auxquels ils font référence

Utilisation HTTP

Verbes HTTP

L'utilisation des verbes HTTP devra respecter la spécification ci-dessous, et, plus généralement, le sens de chaque méthode HTTP tel que décrit dans la section 4.3 de la RFC 7231 (*Table 1*)

Table 1: Utilisation des verbes HTTP pour la construction des APIs

Verbe	Description
PUT	Modification totale d'une ressource
POST	Création une ressource
PATCH	Modification partielle d'une ressource
GET	Récupération d'une ressource
DELETE	Suppression d'une ressource

Entêtes HTTP

Pour chaque réponse retournée, celle-ci doit inclure, à minima :

- La description du format de réponse : Ajout de l'entête **Content-Type**
- La définition de l'encodage utilisé : Ajout de l'entête **Charset**

Code statut HTTP

L'utilisation des codes de retour HTTP devra respecter la spécification suivante, et, plus généralement, le sens de chaque code de retour tel que décrit dans la section 6 de la RFC 7231 (*Table 2*)

Table 2: Utilisation des codes de statut dans les réponses HTTP

Code	Description
2xx	Succès
200	Succès. Des informations de retour sont disponibles.
201	Succès. Une ressource a été créée. Généralement, la réponse contient la ressource qui vient d'être créée.
204	Succès. La réponse ne contient aucune donnée.
4xx	Échec à cause d'un problème dans la requête (exemple : création d'un utilisateur avec un e-mail déjà existant ou paramètre de requête manquant).
5xx	Échec dû à une erreur du serveur

Convention de création pour les API

Afin de maintenir une cohérence forte entre tous les services, certains besoins doivent utiliser une syntaxe commune décrite ci-dessous.

Règles communes

Le résultat de la requête devra toujours être retourné dans le champ **data** d'un objet JSON. Les autres attributs peuvent servir à ajouter des métadonnées à la requête.

API Paginées

Les API paginées acceptent toujours deux paramètres optionnels :

- **page** - Numéro de la page à retourner (défaut : 1)
- **size** - Nombre de résultats par page à retourner (défaut : dépend de l'API, généralement 20) Il doit être possible de manipuler ces paramètres pour obtenir les pages suivantes ou augmenter le nombre de résultat dans une seule page.

API de recherche

Dans le cas d'une API permettant d'effectuer une recherche (Recherche exclusive) sur une ressource :

- Il doit être possible, en spécifiant des valeurs en **query params**, de filtrer les résultats uniquement sur un critère précis de la ressource. *Exemple* : `users/?email=john.doe@contoso.com` - Liste des utilisateurs dont le nom est égal à la valeur indiquée.
- Les arguments de recherches de type string peuvent être préfixés/-suffixés d'un ***** pour rendre la recherche non-limitative. *Exemple* : `users/?username=Sandbob*` - Tous les utilisateurs dont le nom commence par "Sandbob".
- Une logique similaire existe pour les champs de type date, avec les préfixes : **<** et **>**. *Exemple* : `users/?createdAt>=2020-01-15` - Tous les utilisateurs créés après le 15/01/2015.

Il est aussi possible de créer des APIs permettant de sélectionner un ensemble de ressources, correspondant aux différentes valeurs des éléments indiqués dans les query params de la requête HTTP (Recherche inclusive).

- Une syntaxe basée sur des crochets (**[]**) permet de spécifier la liste des différentes valeurs séparées par des virgules (**,**). *Exemple* : `contracts/?id=[1124521,1124550,2102450]` - Obtient une liste des contrats indiqués.

- Une syntaxe supplémentaire peut être implémentée, permettant une sélection sur un range de valeurs, en utilisant le séparateur "...". *Exemple* : `users/?id=[1..5]` (Les utilisateurs dont l'identifiant est contenu entre 1 et 5).

Authenticated API

Certains services API doivent disposer de end-points adaptant leur retour en fonction du contexte d'identité véhiculé à travers le jeton d'authentification. Dans ce scénario, les API doivent répondre aux règles suivantes :

- l'URL contient toujours, juste après le nom du service API et de sa version, le nom de l'identité utilisée. *Exemple* : `/user/contracts` - Les contrats de l'utilisateur xxx.
- Le nom de l'identité doit être au singulier.
- Une authentification de type **Authorization Code** ou **Resource Owner** ↪ **Password Credential** est requise.
- Une erreur 401 (**Unauthorized**) doit être levée si le jeton ne contient pas d'identité ou que celle-ci ne peut pas être vérifiée via le serveur d'autorisation.

Données manipulées par les services

Format d'échange

Les règles suivantes s'appliquent concernant les données des services API :

- Les services API doivent être conçus pour accepter des données d'entrée au format `application/json`.
- Les données retournées par les services API doivent être au format `application/json`.

Par ailleurs, les formats suivants doivent être toujours respectés (en entrée comme en sortie) (*Table 3*).

Table 3: Format d'échanges des données au sein des APIs. *<https://tools.ietf.org/html/rfc3339>

Type de données	Format attendu	Stockage BDD
Dates & heures	Date conforme à la RFC 3339*. <i>Exemple</i> : <code>2005-08-15T15:52:01+01:00</code> <code>DATETIME</code>	

Type de données	Format attendu	Stockage BDD
Chaîne de caractères	Les strings doivent toujours être : - Débarrassées des espaces blancs inutiles (trim) - Utiliser le null si elles sont vides, sauf contrainte métier.	Variable (VARCHAR, TEXT...)
Nombre	Les nombres doivent être représentés sous la forme integer ou float et non de chaîne de caractères.	Variable (INT, NUMERIC...)
Booléen	Les booléens doivent être échangés sous leur forme originelle : true et false . L'utilisation du 0 ou 1 est proscrite.	BIT
Mot de passe	Les mots de passe doivent être hachés en utilisant l'algorithme SHA256	Variable (VARCHAR, TEXT...)

Gestion des Entrées/Sorties

Contrôles d'intégrité des données Pour chaque API, les données d'entrée / sortie doivent être contrôlées sur deux aspects :

- Présence ou non de la donnée.
- Respect du format attendu selon les contraintes de la base de données et selon les règles métiers.

Manipulation des données par les services API Lors de la manipulation des données dans les services API (connexions aux bases de données ...), les règles suivantes devraient être respecter :

- Chaque service est propriétaire de ses données. Il est le seul à pouvoir les consommer. Quelques exceptions peuvent exister pour les données métiers transverses.
- L'utilisation d'un ORM doit être privilégiée pour manipuler les données.
- Les transactions doivent être utilisées chaque fois que nécessaire.
- Les requêtes sollicitant régulièrement la base de données devraient utiliser des connexions persistantes.
- Les chaînes de connexion devraient spécifier selon le modèle l'option de lecture / écriture.

Gestion des erreurs

Normalisation de la sortie d'erreur (API Problem)

Toutes les erreurs API, qu'elles soient techniques ou fonctionnelles, devront être formatées selon les spécifications de la RFC 7807.

Ainsi, une erreur sera toujours conforme, à minima, au format suivant :

```
2 {  
4   "title": "string",  
   "type": "string",  
   "status": int  
}
```

Où :

- **title** : contiendra le nom de l'erreur dans un format lisible par un être humain. (Généralement en anglais)
- **type** : contiendra l'identifiant du type de l'erreur. Deux erreurs ayant la même cause renverront **toujours** un type similaire. Le type peut être utilisé pour permettre d'adapter le retour à afficher à l'utilisateur.
- **status** : Contendra le code HTTP de l'erreur. Sauf mention contraire, ce code sera **toujours** identique à celui de la réponse HTTP.

Par ailleurs, les règles suivantes s'appliquent :

- Les requêtes invalides se traduiront toujours par l'envoi d'une réponse 400 **Bad Request** et d'un type d'erreur : **validation-error**.
- Les requêtes vers des ressources manquantes se traduiront toujours par l'envoi d'une réponse 404 **Not Found** et d'un type d'erreur : **resource-not-found**.
- Si une erreur technique survient durant le traitement, la réponse sera toujours une 500 **Internal Error** et le type d'erreur : **internal-error**.
 - Aucun détail de l'erreur ne devrait être visible en production.
 - L'erreur doit être loguée.

Stockage des erreurs (logs)

Les règles suivantes s'appliquent lorsqu'une erreur survient au sein d'un service API :

- Une solution de log centralisée (ex : **Graylog**) doit être utilisé pour inscrire tous les logs de l'application.
 - Cependant, une application peut disposer **en complément** (duplication) d'un mécanisme de log interne.

- Tous les services API doivent pouvoir activer un mode **debug** (via fichier d'environnement) permettant de loguer toutes les requêtes atteignant le service.
- Les données sensibles devront être anonymisées (exemple : Mot de passe).
- Toute erreur doit être identifiée et doit être traitée sous forme d'exception. Ces exceptions devront toutes être transmises au gestionnaire de log.
- Une requête en erreur doit être, **à minima**, être composée de logs permettant de retrouver :
- Une identification de la cause (détail de l'erreur, cause de l'erreur, fichier, n° de ligne ...)
- Des informations sur la requête ayant générée l'erreur (Données d'entrée, fonction appelée etc. ...)
- Les logs d'erreurs doivent être conservés au moins **30** jours.

Les codes erreurs utilisés pour identifier le niveau de criticité devront être conforme aux recommandations de la section 6.2 de la RFC Syslog Protocol (*Table 4*)

Table 4: Liste des codes erreurs - Syslog Protocol

Code	Gravité	Description
0	Emergency	Système inutilisable.
1	Alert	Une intervention immédiate est nécessaire.
2	Critical	Erreur critique pour le système.
3	Error	Erreur de fonctionnement.
4	Warning	Avertissement (une erreur peut intervenir si aucune action n'est prise).
5	Notice	Événement normal méritant d'être signalé.
6	Informational	Pour information.
7	Debugging	Message de debug.

Version des API

Version d'une API

Chaque service API est **obligatoirement** préfixé d'un numéro de version. Un même service API peut exister sous plusieurs versions.

Un changement de version peut avoir lieu :

- Lors d'une modification du format d'entrées/sorties d'un ou plusieurs end-points constituant un service API.

- Lors de modification profonde des fonctionnalités associées à un ou plusieurs end-points d'un service API, et dont la modification peut avoir un impact sur les applicatifs consommateurs.

Toutes les autres mises à jour (amélioration de performance, correctif de sécurité ...) d'un service API qui n'ont pas d'impact sur les éléments décrit ci-dessus doivent être appliquées de manière transparente sur l'API sans changement de version.

Tenue d'un CHANGELOG

La mise en place d'un **manifeste d'historique de version est obligatoire**.

- Utilisation du format Keep a Changelog & utilisation du Semantic Versioning.
- Ce manifeste doit être présent à la racine du dépôt `Git` sous la forme d'un fichier `CHANGELOG.md` (Fichier reconnu par `Gitlab`)

Documentation API

Pour chaque service, il est nécessaire de fournir :

Une spécification complète de l'API au format Open API 3.0 (Swagger), qui doit inclure à minima :

- Pour chaque **end-point** :
 - Une description claire de son rôle. Cette description doit indiquer si des filtres implicites sont appliqués sur la ressource retournée.
 - Une description des paramètres de recherches s'ils existent.
 - La liste complète des réponses (erreurs / succès) qui peuvent être retournées par la fonction.
 - Pour chaque réponse, un exemple de valeurs retournées et/ou le `model` associé.
- Pour chaque `model` :
 - Une description générale du model.
 - Une description de la signification de chaque champ. Cette description est **obligatoire** même si le nom du champ semble être suffisamment explicite.

Si nécessaire, une documentation contextuelle supplémentaire (par exemple : Liste des règles métiers spécifiques à cette fonction) sera fournie et accessible au même endroit que la documentation **Swagger**.

References

- Divine, Patrick. “String Case Styles: Camel, Pascal, Snake, and Kebab Case.” Medium, July 19, 2019. <https://betterprogramming.pub/string-case-styles-camel-pascal-snake-and-kebab-case-981407998841>.
- Fielding, Roy T., and Julian Reschke. “Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.” Request for Comments. Internet Engineering Task Force, June 2014. <https://doi.org/10.17487/RFC7231>.
- Newman, Chris, and Graham Klyne. “Date and Time on the Internet: Timestamps.” Request for Comments. Internet Engineering Task Force, July 2002. <https://doi.org/10.17487/RFC3339>.
- “Transaction informatique.” In Wikipédia, December 20, 2021. https://fr.wikipedia.org/w/index.php?title=Transaction_informatique&oldid=189066265.
- Nottingham, Mark, and Erik Wilde. “Problem Details for HTTP APIs.” Request for Comments. Internet Engineering Task Force, March 2016. <https://doi.org/10.17487/RFC7807>.
- Gerhards, Rainer. “The Syslog Protocol.” Request for Comments. Internet Engineering Task Force, March 2009. <https://doi.org/10.17487/RFC5424>.
- J. Bernard. “Les principaux formats de flux video live DASH et HLS.” Blog Eleven Labs, July 19, 2017. <https://blog.eleven-labs.com/fr/video-live-dash-hls/>.
- Preston-Werner, Tom. “Semantic Versioning 2.0.0.” Semantic Versioning. Accessed November 27, 2022. <https://semver.org/spec/v2.0.0.html>.
- “CHANGELOG.Md.” Accessed November 27, 2022. <https://changelog.md/>.
- “About Swagger Specification | Documentation | Swagger.” Accessed November 27, 2022. <https://swagger.io/docs/specification/about/>.

Backend For Frontend : Notes

LeFrançois, Marc

Le-Point-Technique, January/2023

abstract: L'utilisation d'une couche ou application BFF (Backend For Frontend) est utile lorsque le système possède des interfaces différentes, afin d'avoir un rendu de données spécifique en fonction du besoin de chacune desdites interfaces. Cela présente notamment l'avantage d'économiser de la bande passante via l'optimisation des requêtes suivants le besoin ; c'est le travail que fait le BFF.

keywords: Backend For Frontend (BFF), Pattern, Software Architecture

Avantages

- Une séparation nette des responsabilités entre développeur backend et frontend :
 - Les développeurs backend mettront à disposition des API “générique” et contenant des données brutes.
 - Tandis que les développeurs frontend pourront spécifier leurs besoins de données en fonction des interfaces.
- Optimisation des requêtes selon les besoins.
- La maintenance corrective spécifiques pour chaque BFF/interfaces.
- L'isolation de chaque interface et leur BFF associé
- Les développeurs frontend peuvent faire évoluer les requêtes suivants leur convenance.
- Le backend quant à lui fourni une API qui est indépendante des interfaces.
- Certaines données sensibles peuvent être cachées à des clients spécifiques.

Inconvénients

- Une couche et un service supplémentaire à déployer, maintenir et superviser.
- L'ajout d'une couche peut engendrer une augmentation légère de la latence.
- Les développeurs frontend doivent gérer en supplément un backend.
- Duplication de code.

Implémentation

La *Figure 1* contient un schéma générique d'une implémentation BFF.

Afin de pallier du mieux possible aux inconvénients vus plus haut, une alternative est la création d'un BFF unique par interface et pour plusieurs entrées (p.ex. site Web et l'application mobile), notamment via l'utilisation de la solution GraphQL, qui permet de n'avoir qu'une seule couche BFF pour des entrées multiples.

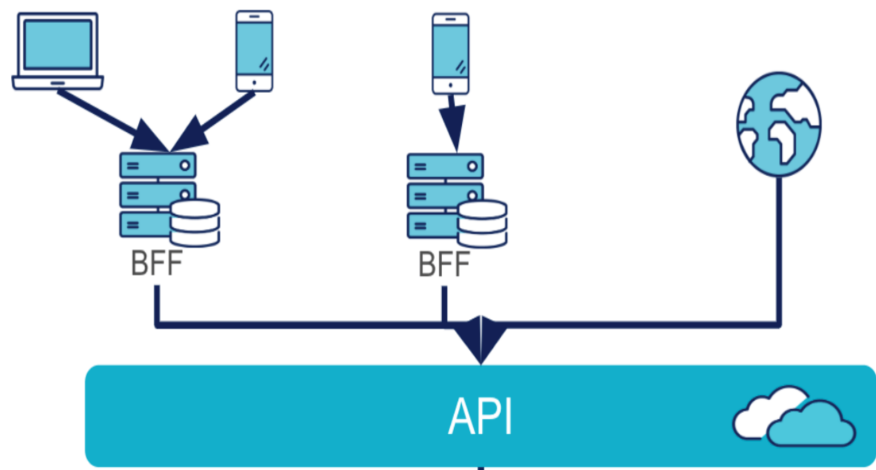


Figure 1: Schéma générique d'un BFF.

Une seule interface connectée à un unique BFF, et accessible via plusieurs entrées ou terminaux (fixe et mobile), permettrait (en plus des avantages déjà susmentionnés): - de limiter la duplication de code, de la standardiser notamment avec du HTML 5 ; - de limiter la latence, limitant le nombre d'interfaces et donc de BFFs ; - de faciliter la maintenance, pour les mêmes raisons susmentionnées, et de fait d'alléger la charge de travail des développeurs frontend.

Référence

A. Bhayani, 'BFF - Backend for Frontend - Pattern in Microservices | LinkedIn'. <https://www.linkedin.com/pulse/bff-backend-frontend-pattern-microservices-arjit-bhayani/>.

P. Trollé, 'Les indispensables d'un projet frontend - Un Backend For Frontend, une API sur-mesure', OCTO Talks!, Jan. 11, 2019. <https://blog.octo.com/les-indispensables-dun-projet-frontend-un-backend-for-frontend-une-api-sur-mesure/>.

R. Calamier, 'GraphQL: Et pour quoi faire?', OCTO Talks!, Aug. 09, 2018. <https://blog.octo.com/graphql-et-pourquoi-faire/>.

Notes on Large Language Models

Andreev, Anton

Le-Point-Technique, June/2023

abstract: In this technical brief, we review some common questioning about Large Language Models (LLM), such as: What are the types of LLM? What is fine-tuning? or What is prompting?

keywords: LLM, Prompting, Fine-tuning

Common applications of LLM

Common applications of LLM can be separated in two domains: dialog and natural language processing (NLP) tasks.

In terms of dialog, applications regroup: - task-oriented dialog - conversational question answering - open domain dialog

In terms of NLP tasks, applications regroup: - sentence classification - named entity recognition - summarization - masked language modeling - many more ...

Popular LLM are ChatGPT (OpenAI, San Francisco, the US), LLaMa (Meta, Menlo Park, the US) or again Hugging Chat (Hugging Face, New York City, the US). You can find a list of LLMs [here](#).

What are the types of language modelling?

- *Masked language modeling* (MLM) predicts a masked token in a sequence, and the model can attend to tokens bidirectionally. This means the model has full access to the tokens on the left and right. Masked language modeling is great for tasks that require a good contextual understanding of an entire sequence. MLM is used by bi-directional models like BERT, in which a certain percentage of words in the training set are masked, and the task of the model is to predict these missing words. Note that in this task, the model can see the words preceding as well as succeeding the missing word and that's why its called bi-directional. For pre-training BERT, another task called Next Sentence Prediction (NSP) was also used, but researchers have found its utility to be marginal and MLM being good enough for all practical purposes.
- *Causal language models* (CLM) are frequently used for text generation. You can use these models for creative applications like choosing your own text adventure or an intelligent coding assistant like Copilot or CodeParrot. Causal language modeling predicts the next token in a sequence of tokens, and the model can only attend to tokens on the left. This means the model cannot see future tokens. GPT-2 is an example of a causal language model.

- An example of auto-regressive model is GPT. These models are uni-directional (one directional) and they are trained to predict the next word without seeing the succeeding ones. This is because these auto-regressive models are specifically designed for better language generation, which makes it necessary for the model to be pre-trained in a uni-directional manner.

Comparison of the number of parameters in both types of models: - BERT with a few hundred million parameters (MLM). - GPT-like models which have several billion parameters (AR).

What is PLM?

“PLM” stands for pre-trained language models, as compared to from-scratch model. We will review here some technics to use these pre-trained models.

What is fine-tuning?

It is also commonly referred to as *transfer learning*. There are two types of fine-tuning: *task oriented* and *domain adaptation*:

- For many NLP applications involving Transformer models, you can simply take a pretrained model from the Hugging Face Hub and fine-tune it directly on your data for the task at hand. Provided that the corpus used for pretraining is not too different from the corpus used for fine-tuning, transfer learning will usually produce good results.
- However, there are a few cases where you will want to first fine-tune the language models on your data, before training a task-specific head. For example, if your dataset contains legal contracts or scientific articles, a vanilla Transformer model like BERT will typically treat the domain-specific words in your corpus as rare tokens, and the resulting performance may be less than satisfactory. By fine-tuning the language model on in-domain data you can boost the performance. This process of fine-tuning a pretrained language model on in-domain data is usually called domain adaptation.

What is prompting?

Prompt is where you can describe to the model what you want it to do. It like giving somebody an instruction. For example, we can give the instruction “Write a sentence using the word ocean” and it can respond with: “The ocean is vast and beautiful.”

Prompting can be separated 3 categories (although not necessarily): - instruction: “Write a sentence”, “Write a resume on the subject of . . .” - examples for few-shot learning - the model will take cues from the writing style and try to use the

same style of writing - context stuffing - these are specific details such as name, materials or places to be used in the output generation

At its base LLM do a word prediction trying to complete a phrase. This means that even the instructions we give can be taken as a text to be completed instead of producing the desired result. In such case you need to specify the instruction more clearly.

An example using Cohere Playground:

```
Name: Peter
2 Age: 38
Occupation: doctor
4 Description: Answer the question "What is your name?"
```

And the output is: My name is Peter.

Note how the first three lines are used to provide a context and the last line is the instruction.

What is in-context learning (few-shot learning)?

In-context or few-shot learning (FSL) through prompt design is the process of fine-tuning a model with a hundred or thousand input texts, the model just takes a few task-specific examples (<10 usually) as input, and then quickly figures out how to perform well on that task. Note that in this process, there is no update of the model weight that happens! No backpropagation and no gradient descent! Researchers have suggested that GPT-like models do some kind of Bayesian inference using the prompts. In simple words, using just a few examples, the model is able to figure out the context of the task, and then it makes the prediction using this information.

Few-Shot Learning (FSL) is a Machine Learning framework that enables a pre-trained model to generalize over new categories of data (that the pre-trained model has not seen during training) using only a few labeled samples per class.

Few-Shot learning (LSL) is where we prompt the model with a few examples so that it picks up on the pattern and style we are going for.

What is RLHF training?

At a high-level, reinforcement learning with human feedback (RLHF) is a technique for training large language models that has been critical to OpenAI's ChatGPT and InstructGPT models, Anthropic's Claude, and more. RLHF enables training LLMs to understand instructions and generate helpful responses.

What is federated learning?

Under federated learning, multiple people remotely share their data to collaboratively train a single deep learning model, improving on it iteratively, like a team

presentation or report. Each party downloads the model from a datacenter in the cloud, usually a pre-trained foundation model. They train it on their private data, then summarize and encrypt the model new configuration. The model updates are sent back to the cloud, decrypted, averaged, and integrated into the centralized model. Iteration after iteration, the collaborative training continues until the model is fully trained.

Classification of LLMs

LLM can be regrouped in two categories:

- Grounded - is the linking of concepts to context and within NLP context is often a knowledge base, images or conversation
- Instruct GPT - is a powerful tool that allows users to fine-tune the language generation capabilities of the GPT (Generative Pre-trained Transformer) model. Developed by OpenAI, Instruct GPT allows users to train the model on specific tasks and generate text that is tailored to their specific needs.

– Example:

```
Correct spelling and grammar from the following text.  
2 I do not wan to go
```

References

A. Andreev, 'List of Large Language Models and APIs'. https://github.com/toncho11/ML_examples/wiki/List-of-Large-Language-Models-and-APIs.

'What are foundation models? | IBM Research Blog'. <https://research.ibm.com/blog/what-are-foundation-models>.

Mentions Légales

Titre de la revue : Le-Point-Technique

Éditeur de la publication : Grégoire Cattan, Montpellier, France

ISSN : 2826-5726