

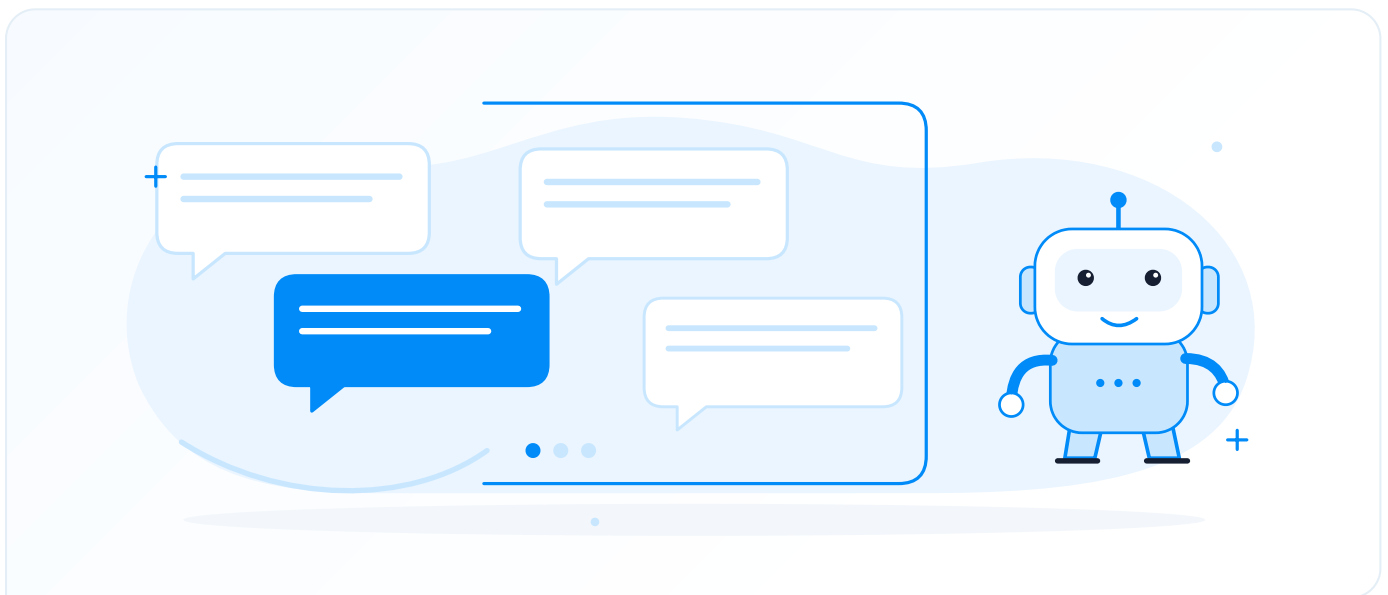
Tutoriel

LLM · Chatbot · API

Construire un chatbot avec mémoire courte

Construire une boucle de conversation minimale avec historique borné, commande de remise à zéro et journalisation locale.

Auteur **Youness Boutyour**

**DURÉE**

50 minutes

NIVEAU

Débutant avancé

FORMAT

Tutoriel pratique

LLM · Chatbot · API · publié le 2026-06-02 · mock

Objectifs et livrables

Un chatbot doit pouvoir suivre une conversation : une précision apportée au deuxième message peut changer le sens du troisième. Nous allons construire un assistant qui conserve les échanges récents tout en gardant une mémoire de taille limitée.

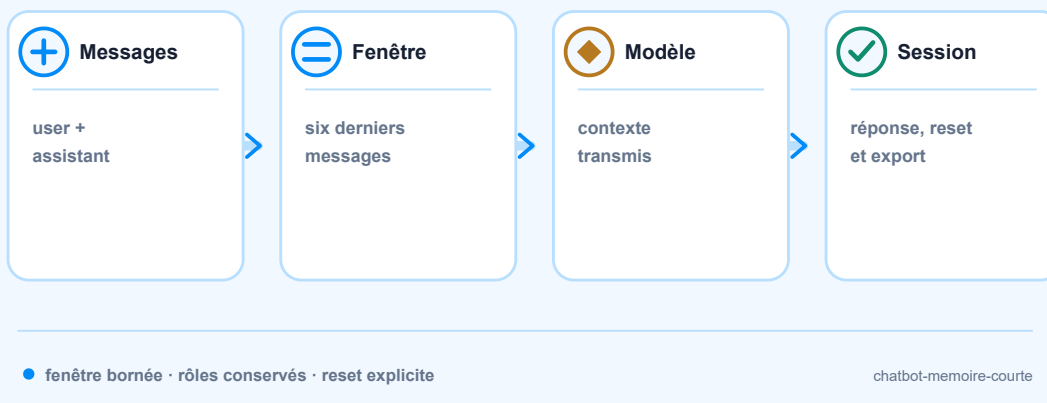
Le travail commence par la représentation des messages et de leurs rôles, puis par la construction d'une fenêtre de contexte. Nous ajouterons l'appel au modèle et suivrons plusieurs tours pour observer les informations transmises à chaque demande. Vous verrez notamment ce qui se passe lorsque les premiers messages quittent la fenêtre.

Nous intégrerons aussi la commande `/reset` pour repartir sur une conversation vide. Elle sera traitée par le programme, et vous pourrez vérifier l'état de la session après son exécution. Ces essais aideront à comprendre le lien entre la continuité du dialogue, le volume de contexte et les données conservées.

Le fichier `outputs/session.json` rendra la mémoire consultable. À la fin du tutoriel, vous pourrez relire les échanges, contrôler la limite choisie et vérifier que la réinitialisation produit bien une nouvelle session.

Faire varier une fenêtre de contexte

Schéma du chatbot qui borne les messages transmis au modèle, accepte `/reset` et exporte l'état observable.



La mémoire utile est bornée : elle conserve le contexte récent sans transmettre toute la conversation.

Résultat du chatbot à mémoire courte

Une boucle locale, six messages conservés et une session réinitialisable.

The image shows a terminal window on the left and a context window on the right. The terminal window, titled 'Terminal - Chatbot mémoire courte', displays the following commands and outputs:

```
$ python src/chat.py --mock --demo
history_size=6

$ python src/chat.py --mock --reset
history_size=0

✓ session écrite dans outputs/session.json

fenêtre active : 6 messages maximum

$ python src/chat.py --mock --question
"Quel est mon sujet ?"
```

The context window, titled 'Fenêtre de contexte', shows the state of the chatbot's memory. It indicates that the last six messages are transmitted. A button labeled 'limit = 6 · mémoire bornée' is present. Below it, a list of messages is shown, each with a role (user or assistant) and a status (detail, previous, context, recent, or generated). At the bottom, a button labeled '/reset' is shown, with a note that it clears the memory and resets to zero.

Résultat obtenu : un chatbot en ligne de commande qui conserve les six derniers messages et exporte la session.

Lire le flux avant de coder

Le schéma suit la construction du contexte envoyé à chaque tour. Les messages sont conservés avec leur rôle, puis une fenêtre bornée retire les plus anciens avant l'appel du modèle. La commande `/reset` agit sur la session complète et ne doit pas être confondue avec une simple réponse textuelle.

Pour comprendre le comportement, observez la taille de la fenêtre après chaque message. Une information oubliée n'est pas forcément une erreur du modèle : elle peut avoir quitté le contexte transmis. L'export permet de distinguer l'état conservé localement de ce qui a réellement été envoyé au fournisseur.

1. Préparer l'environnement

Avant un premier essai, choisissez si vous repartez d'une session vide ou si vous rechargez le fichier existant. Cette décision change le contexte envoyé au modèle, même avec une question identique. Les commandes restent exécutées depuis la racine du projet pour que le chargement et l'export utilisent le même `outputs/session.json`, plutôt que deux fichiers créés dans des dossiers différents.

La session sera stockée dans `outputs/session.json`. Le chargement vérifie d'abord l'existence du fichier pour distinguer une première conversation d'une reprise. Il accepte les deux formes montrées dans le projet : une liste de messages ou un objet contenant la clé `messages`.

Observez le résultat de ce chargement avant le premier appel au modèle. Une session absente doit donner une liste vide, tandis qu'un fichier existant doit restituer les rôles et les contenus déjà sauvegardés. Un fichier JSON invalide relève d'un problème de persistance à corriger avant d'examiner le comportement du chatbot.

 Terminal · Bash

```
mkdir -p chatbot-memoire-courte/{src,outputs}
cd chatbot-memoire-courte
python3 -m venv .venv && source .venv/bin/activate
python -m pip install requests
```

Sortie obtenue

 Sortie / exemple

```
Successfully installed requests
```

Commande de vérification

 Terminal · Bash

```
python -c "import requests; print('requests ok!')"
```

Code de l'étape

La mémoire est conservée dans un fichier JSON afin de pouvoir reprendre une session :

 Python

```
import json
from pathlib import Path

history_path = Path("outputs/session.json")
history = []
if history_path.exists():
    stored = json.loads(history_path.read_text(encoding="utf-8"))
    history = stored.get("messages", []) if isinstance(stored, dict) else stored
```

2. Définir la mémoire bornée

Suivez la valeur renvoyée par `remember` et réaffectez-la à l'historique lorsque vous utilisez la version qui construit une nouvelle liste. Appeler la fonction sans conserver son retour laisserait l'ancienne mémoire en place. La limite porte sur les messages conservés, pas sur leur longueur : six messages très longs peuvent encore produire une requête volumineuse, et le message système est assemblé séparément.

La limite porte ici sur le nombre de messages, et non sur le nombre de tokens. Avec `limit=6`, la fonction conserve au plus six objets après l'ajout du nouveau message. Une question et sa réponse représentent deux messages ; un texte très long peut donc toujours produire un contexte volumineux malgré cette limite.

Les deux versions de `remember` illustrent la même fenêtre, mais la seconde construit une nouvelle liste au lieu de modifier l'ancienne. Utilisez une seule convention dans votre programme. La remise à zéro doit être traitée avant l'ajout de la question : le texte `/reset` ne doit pas devenir un message destiné au modèle.

Dans `src/chat.py`, utilisez une liste et une fonction explicite :

● ● ● Python

```
def remember(history, role, content, limit=6):
    history.append({"role": role, "content": content})
    return history[-limit:]
```

Ajoutez `/reset` avant de transmettre l'historique au modèle.

Code de l'étape

● ● ● Python

```
def remember(history, role, content, limit=6):
    messages = history + [{"role": role, "content": content}]
    return messages[-limit:]

if reset_requested:
    history = []
else:
    history = remember(history, "user", question)
```

Sortie obtenue

Après huit messages, le contexte transmis ne contient que les six derniers.

■ Sortie / exemple

```
turns=8 history_sent=6 oldest_role=user newest_role=assistant
```

Commande de vérification

Terminal · Bash

```
python src/chat.py --mock --demo | tee outputs/session.txt  
grep -q 'history_size=6' outputs/session.txt
```

3. Ajouter l'appel au modèle

Repérez l'endroit exact où la question courante entre dans la requête : elle doit y apparaître une seule fois. Les extraits montrent les éléments du mécanisme, mais l'historique déjà enrichi ne doit pas recevoir à nouveau le même message pendant l'assemblage. Après l'appel, enregistrez la réponse de l'assistant pour que le prochain échange retrouve les deux côtés de la conversation.

Le payload combine une consigne système, les échanges récents et la nouvelle question. Vérifiez cet assemblage avant l'envoi : la question courante doit apparaître une seule fois. Si vous l'avez déjà ajoutée à `history` dans l'étape précédente, ne la rajoutez pas une seconde fois dans la liste transmise.

Après réception de la réponse, enregistrez le tour utilisateur puis le tour assistant avec la même règle de troncature. Le mock permet de contrôler ce mécanisme indépendamment du réseau. Comparez ensuite le mode réel sur la même séquence de questions, en regardant quelles informations quittent la fenêtre et à partir de quel tour elles ne sont plus disponibles.

Construisez un payload avec un message système, l'historique et le dernier message. En mode `--mock`, retournez une réponse déterministe pour isoler la logique de mémoire.

Sortie obtenue

La boucle affiche une réponse après chaque question et n'envoie jamais plus de six messages d'historique.

Sortie / exemple

```
user: Quel est mon sujet ?  
assistant: Votre sujet est la qualité des données.  
history_size=6
```

Avec un endpoint compatible configuré par `GENAI_BASE_URL` et `GENAI_MODEL`, lancez `python src/chat.py --real --question "Quel est mon sujet ?"` pour remplacer la réponse mock par une réponse générée.

Code de l'étape

Le modèle reçoit le système, l'historique borné et la nouvelle question :

Python

```
from llm_client import chat

messages = [
    {"role": "system", "content": "Réponds brièvement et clairement."},
    *history,
    {"role": "user", "content": question},
]
response = chat(messages)
history = remember(history, "user", question)
history = remember(history, "assistant", response)
```

Commande de vérification

Terminal · Bash

```
python src/chat.py --mock --question "Quel est mon sujet ?"
```

La figure « Résultat du chatbot » montre le livrable tel qu'il est exécuté dans le terminal : une question, une réponse, une fenêtre de six messages et les contrôles `/reset` et `outputs/session.json`. Elle représente le mode mock reproductible ; le mode réel conserve exactement la même structure de mémoire.

4. Exporter et tester la session

Pour tester l'oubli, posez une question qui dépend d'un détail ancien, ajoutez suffisamment d'échanges pour le sortir de la fenêtre, puis reposez la question. Vérifiez d'abord les messages réellement envoyés, avant d'interpréter la réponse du modèle. Réinitialiser une variable en mémoire ne supprime pas automatiquement la session persistée ; le prochain lancement peut la recharger si le fichier reste inchangé.

La sauvegarde écrit un objet contenant les messages et la limite appliquée. Le fichier devient ainsi un moyen de vérifier l'état de la conversation après l'arrêt du programme. L'appel à `mkdir` prépare le répertoire de sortie et l'encodage UTF-8 conserve correctement les questions en français.

Effectuez les contrôles dans un ordre simple : envoyer quelques questions, relire le JSON, atteindre la limite, puis demander la réinitialisation. Après cette dernière opération, redémarrez aussi le programme pour vérifier qu'un ancien historique n'est pas rechargé depuis le disque. Une mémoire vide uniquement à l'écran ne suffit pas à démontrer que la session a été réinitialisée.

Écrivez `outputs/session.json` et testez `/reset`, l'historique borné et le comportement d'une question vide. Une entrée vide doit être refusée localement.

Sortie obtenue

Le fichier JSON contient les rôles, les contenus et le nombre de tours ; `/reset` produit une mémoire vide.

Sortie / exemple

```
export=outputs/session.json turns=8 reset_history_size=0
```

Commande de vérification

Terminal · Bash

```
python -m json.tool outputs/session.json >/dev/null
python src/chat.py --mock --reset | grep -q 'history_size=0'
```

Code de l'étape

Écrivez la session après chaque tour et contrôlez la taille réellement conservée :

Python

```
history_path.parent.mkdir(exist_ok=True)
history_path.write_text(
    json.dumps({"messages": history, "limit": 6}, ensure_ascii=False, indent=2),
    encoding="utf-8",
)
assert len(history) ≤ 6
```

Exemples de sorties

Le mode mock fonctionne sans réseau, respecte la limite de six messages et exporte une session lisible. Avec un vrai modèle, comparez la latence et le nombre de tokens.

Dépannage et diagnostics

Ne placez jamais la clé API dans le code. Utilisez une variable d'environnement et masquez les prompts dans les logs si la session contient des données personnelles.

Vérification finale

- ☐ mémoire limitée à six messages ;

- ☐ `/reset` testé ;
- ☐ question vide refusée ;
- ☐ session exportée sans secret.

Limites et prolongements

Une mémoire courte perd les faits anciens. Un prochain épisode peut ajouter un résumé contrôlé, mais celui-ci devra être testé comme une nouvelle sortie générée.

Références

- [OpenAI — Text generation guide](#)
- [Python — os.environ](#)