# A Quick Developer Guide To Incorporating ChatGPT Into Jakarta EE Applications

The Payara® Platform - Production-Ready, Cloud Native and Aggressively Compatible.

User Guide

## Contents

Guide Updated: **July 2023**

Artificial Intelligence (AI) has emerged as a transformative technology with a profound impact on numerous domains. Healthcare, manufacturing, retail, education and banking and finance are all areas that have been impacted and continue to be impacted by the rise of AI. The rapid advancement in the field has created opportunities for developers to tap into, whether you are building a pet or enterprise project.

Arguably the most popular AI tool out there at this point in time is OpenAI's ChatGPT. ChatGPT, with the GPT standing for Generative Pretrained Transformer, is an advanced language model, based on GPT-3.5 architecture, that is designed to generate human-like text responses given an input prompt or a series of prompts.

ChatGPT has been trained on a diverse range of internet text data to identify patterns, grammar, and context. It is capable of understanding coherent and contextually relevant conversations across different fields of knowledge. Trained on a massive scale, the model can be used for a variety of purposes, including chatbot applications, content generation, language translation among others.

In this quick guide, you will learn how to incorporate ChatGPT into your Jakarta EE applications using the OpenAI GPT API. We start off by taking a brief look at the OpenAI API, then assess the most optimal way to consume it in a typical Jakarta EE application. Finally, we dive into the code and see how you can structure your code to incorporate the features of ChatGPT into your application. By the end of this guide, you will have an overview of how to take advantage of the OpenAI service in your Jakarta EE application.

Before we get started, look at the app from the sample code used in this guide, deployed to Payara Cloud. It is a pure Jakarta EE application that takes a city, and a budget, then returns a list of places with budgeted suggestions. Payara Cloud is a modern, easy to use and fully automated Jakarta EE/ MicroProfile cloud deployment service that takes away the need to deploy your own infrastructure.

## OpenAI

Founded in December 2015 by Elon Musk, Sam Altman, Greg Brockman, Ilya Sutskever, Wojciech Zaremba, and others, OpenAI is an AI research laboratory and company that "aims to ensure that artificial general intelligence (AGI) benefits all of humanity." It conducts research in machine learning and develops advanced AI models and technologies. It has pushed the boundaries of AI research with the development of the GPT (Generative Pre-trained Transformer) series of language models, which includes models like GPT-3, GPT-3.5 and GPT-4.

OpenAI has a number of products including Dalle, ChatGPT and Whisper. It also has an API service that users can subscribe to incorporate their language models into applications. That is the subject of this guide.

## The GPT API

OpenAI's GPT API is the service that makes available to developers the same large language model (LLM) that powers its ChatGPT application. The GPT models available through the API are 3, 3.5 and 4. GPT 4 is the latest and “most capable” model, yet. It has just been made generally available to all existing subscribers “with a history of successful payments.” This guide uses GPT-3.5.

You can incorporate OpenAI’s GPT models into your application through subscription to their API service. After signing up, you can generate your API key for authenticating to the service. You have an API key, now what are your options in how to consume the service in your Jakarta EE application?

There are several ways you can access the API service. OpenAI itself provides official Python and Node.js client libraries. You can either use the HTTP client that comes with Java SE or use the Jakarta REST Client to access the API from your Jakarta EE application. But that would require some plumbing and extensive understanding of the API to be able to make optimal use of it in a non-trivial application. Microsoft also does offer an Azure OpenAI client library for Java.

## OpenAI Java

However, my personal recommendation is the third-party library, OpenAI-Java. It is a very straight-forward Java library for using OpenAI's GPT APIs. It currently has support for GPT versions 3, 3.5 and 4. To get started with OpenAI-Java, we first add the latest version of the dependency in our pom.xml file as shown below.

```
<dependency>
            <groupId>com.theokanning.openai-gpt3-java</groupId>
            <artifactId>service</artifactId>
            <version>0.14.0</version>
        </dependency>
```

With that in place, you are ready to make calls to API service using your token.

# The Problem Domain

For this guide, you will build a simple application that takes a city name, and a given budgetary amount, then relays that data to the GPT model, along with a “system message” that tells the model what is expected of it given the input data from the user. The system message is detailed enough, complete with the structure and format of the response. The app then takes this response and shows it on the UI. We are not going to delve into every part of the application as the code will be available for you to reference. We will only focus on the parts that concern the subject of this guide.

## The OpenAiService Entry Point

The entry point to using Openai-Java is through the OpenAiService class. This class allows customisations through four constructors. There are different ways you can instantiate the class. However, as the Jakarta Contexts and Dependency Injection API is the standard IoC container on the Jakarta EE Platform, we take advantage of its producer mechanism to have a central producer “factory” for producing OpenAiService objects in our application. This way, we can centralise the instantiation to a single place, and thus have all customisations done at one place.

The class below shows a Jakarta CDI producer method that we use to instantiate an instance of the OpenAiService class.

```
@ApplicationScoped
public class OpenAIFactory {
      @Inject
      @ConfigProperty(name = "open.api.key")
      private String apiKey;
      @Inject
      @ConfigProperty(name = "openai.timeout")
      private int apiTimeout;
      @Produces
      @Singleton
      public OpenAiService produceService() {
            return new OpenAiService(apiKey,
            Duration.ofSeconds(apiTimeout));
      }
}
```

The OpenAiFactory class is a `jakarta.enterprise.context.ApplicationScoped` class that will have only one instance across the lifecycle of the application. The `produceService()` method is annotated `@jakarta.enterprise.inject.Produces` and `@jakarta.inject.Singleton`. The first annotation tells the CDI runtime to invoke this method whenever it encounters a request for an instance of the returned type of the method, in this case the OpenAiService. The second method sets the scope of the returned object. We use singleton as we want only a single instance at any point in time. But wait, you might be saying, did we not just use `@ApplicationScoped` to denote a singleton?

Yes. Both `@ApplicationScoped` and `@Singleton` are very similar in the sense that they both result in a single instance of the annotated class at any point in time. The difference lies in the actual object returned to the client. Whereas `@ApplicationScoped` returns a client proxy, `@SIngleton` returns a real unproxied instance. Though for our use this subtle difference is irrelevant, it's important to keep in mind in your own code.

Fine, but why use different annotations then? The reason for picking `@Singleton` in this example is because the OpenAiService class has no default no-arg constructor nor at least one constructor annotated with `@Inject`. The requirement for a no-arg constructor or at least one constructor annotated `@Inject` is to allow the container to create proxies of the actual objects. If you annotated the `prodcueService()` method with `@ApplicationScoped` for instance, the application will fail deployment with the exception

```
org.glassfish.deployment.common.DeploymentException: CDI deployment
failure:WELD-001410: The injection point has non-proxyable dependencies
```

The OpenAIFactory also has two injected fields, `apiKey` and `apiTimeout`. These fields are used to customise the OpenAiService. The apiKey refers to the API key obtained from OpenAI. You can find this on your account dashboard. The common annotation on both fields, `@org.eclipse.microprofile.config.inject.ConfigProperty` is from the MicroProfile project. This annotation is used to read configuration property from various configuration sources. The properties can be set as environment variables or bundled with the application through the microprofile-config.properties file.

As the API key is security related, it's recommended to set it outside of the application, perhaps as environment variable through

```
export OPEN_API_KEY=sk-htreVoraWAWTsw7hhhcKT3BlbkFJg569alYS60SaVVUJHJopi
```

This way, the MicroProfile config runtime will read it at application runtime and assign the value to the injected field. The apiTimeout is used to set the timeout during the initial connection to the OpenAI service. The default, if none is passed, is 10. With the OpenAiService producer in place, we can inject it into its dependents with the @Inject annotation, much like any other application object.

## The TripAdvisorService

The following class shows the skeleton of the TripAdvisorService, the actual workhorse of the application.

```
@ApplicationScoped
public class TripsAdvisorService {
    @Inject
    private OpenAiService openAiService;
 @Inject
    Cache<Integer, PointsOfInterestResponse> cache;
}
```

### Cost Reduction With Caching

The class has two fields, the OpenAiService and a javax.cache.Cache. The first field is the OpenAiService entrypoint that we inject, to be fulfilled by our produceService producer method in the afore-discussed OpenAIFactory class. The second one is the standard cache API on the Jakarta Platform. As JCache is not part of the core Jakarta EE Platform, we add it explicitly as a dependency in our pom.xml file as follows.

```
<dependency>
          <groupId>javax.cache</groupId>
          <artifactId>cache-api</artifactId>
          <version>1.1.1</version>
     </dependency>
```

The Jcache (JSR107) API is a standard cache that requires an implementation, much like all Jakarta specs. The Payara Platform ships with Hazelcast, a popular implementation of JCahce. As this application runs on the Payara Platform locally and Payara Cloud, we only need to add the API dependency like we did above. The runtime Payara runtime makes the bundled Hazelcast available to our application.

Why do we need a cache? As the OpenAI service is charged per token, or roughly, words, the less trips we make to the model, the cheaper for us. So when a user makes a query, we cache the request and response for a while. If the same query, in this case, the same city and budget amount is requested again, we return the cached data, instead of making a trip to the model which will needlessly count against our consumed tokens. In the above example, we created a typed cache by injecting the `javax.cache.Cache`, giving it a type, in this case the key of the cache being an integer and the value being `PointsOfInterestResponse`, an application artefact.

## Chat Completions

The GPT model uses the concept of chat completions to interact with it. A chat completion is a collection of messages, where each element in the message has a role. The available roles are system, user and assistant. The system message is used to set the behaviour of the model. A typical system message could be as simple as "You are a helpful assistant." A a user message is the actual instruction that the model will execute, for example "who won the world series in 2020?" The system and user messages are then relayed together to the model for a response. The system message for our application, shown below, uses Java 15 text blocks to create a preamble for the model.

```
private static final String SYSTEM_TASK_MESSAGE = """
            You are an API server that responds in a JSON format.

          Don't say anything else. Respond only with the JSON.

            The user will provide you with a city name and available budget.
Considering the budget limit, you must suggest a list of places to visit.

            Allocate 30% of the budget to restaurants and bars.

            Allocate another 30% to shows, amusement parks, and other
sightseeing.

            And dedicate the remainder of the budget to shopping. Remember, the
user must spend 90-100% of the budget. Do NOT go above 100% of the budget.

            Respond in a JSON format, including an array named 'places'. Each
item of the array is another JSON object that includes 'place_name' as a text,

            'place_short_info' as a text, and 'place_visit_cost' as a number.

            Don't add anything else in the end after you respond with the JSON.

            """;
```

As you can see, the system message spans several paragraphs with not-so-few words. This system message, together with the user input in the form of a city and budget will be relayed to the model.

## Calling the GPT Model

With our system message set, we are ready to talk to the GPT model. The method susgestPointsOf-Interest() method shown below is used by the application UI to talk to the model.

```
public PointsOfInterestResponse suggestPointsOfInterest(String city,
BigDecimal budget) {
    int cacheKey = generateKey(city, budget);
    if (cache.containsKey(cacheKey)) {
    return cache.get(cacheKey);
    }
    try {
    String request = String.format(Locale.ENGLISH, "I want to visit %s and
have a budget of %,.2f dollars",
    city, budget);
    var poi = sendMessage(request);

    List<PointOfInterest> poiList = generaPointsOfInterest(poi);

    PointsOfInterestResponse response = new PointsOfInterestResponse();
    response.setPointsOfInterest(poiList);
    cache.put(cacheKey, response);
    return response;
    } catch (Exception e) {
    e.printStackTrace();
    PointsOfInterestResponse response = new PointsOfInterestResponse();
    response.setError(e.getMessage());


    return response;
    }
    }
```

The method takes a String and BigDecimal objects, representing the city and monetary budget as entered by our application user. The method first generates a cacheKey using the two method parameters. The cache key is simply the sum of the hashcode of both the city and budget variables. The cache key is then used to check if the given request is available in the cache. If it is, then the cache value is returned. If not, the method proceeds to format the user message, using the method parameters. The formatted user message is then passed to the sendMessage() helper method.

The `sendMessage()` helper returns a String. This string is a JSON representation of the expected result, in the format in which the model was instructed to put in in the system message (shown earlier). The `generatePointsOfInterest()` method method transforms the given plain JSON string to a typed List of PointOfInterest objects. This list is then placed in the PointsOfInterestResponse wrapper class, saved in the cache, then returned to the client.

The sendMessage() helper method is shown below.

```
private String sendMessage(String message) {
    ChatCompletionRequest chatCompletionRequest = ChatCompletionRequest
    .builder()
    .model("gpt-3.5-turbo")
    .temperature(0.8)
    .messages(
    List.of(
    new ChatMessage("system", SYSTEM_TASK_MESSAGE),
    new ChatMessage("user", message)))
    .build();
    StringBuilder builder = new StringBuilder();
    ChatCompletionResult chatCompletion = openAiService.
createChatCompletion(chatCompletionRequest);


    chatCompletion.getChoices().forEach(choice -> builder.append(choice.
getMessage().getContent()));


    return builder.toString();
    }
```

The `sendMessage()` method is the place where the external API call is made. Using the OpenAi-Java library constructs, a `com.theokanning.openai.completion.chat.ChatCompletionRequest` object is created. This class has a builder that we use to construct the object, passing in the GPT model version and temperature to use. Temperature refers to the randomness and subsequently the creativity of the responses. It is always a number between 0 and 1.

With the ChatCompletionRequest object, we call the OpenAI service by invoking the `createChatCompletion()` method on the injected OpenAiService instance, passing in the chatCompletionRequest object. The method returns a `com.theokanning.openai.completion.chat.ChatCompletionResult` that has a list of `com.theokanning.openai.completion.chat.ChatCompletionChoice`. The ChatCompletionChoice class has a message field that we then call during each iteration, put into a String and return.

The returned string from the sendMessage() method is transformed into concrete Java types using the Jakarta JSON-P API in the generaPointsOfInterest() method shown below.

```
private List<PointOfInterest> generaPointsOfInterest(String json) {
    try (JsonReader reader = Json.createReader(new StringReader(json))) {


    JsonObject jsonObjectResponse = reader.readObject();
    JsonArray placesArray = jsonObjectResponse.getJsonArray("places");


    List<PointOfInterest> poiList = new ArrayList<>(placesArray.size());


    for (int i = 0; i < placesArray.size(); i++) {
    JsonObject jsonObject = placesArray.getJsonObject(i);
    PointOfInterest poi = PointOfInterest
    .builder()
    .info(jsonObject.getString("place_short_info"))
    .cost(BigDecimal.valueOf(jsonObject.getInt("place_visit_cost")))
    .name(jsonObject.getString("place_name"))
    .build();


    poiList.add(poi);
    }


    return poiList;
    }
    }
```

The method reads in the JSON string, then given the structure, the various information is fetched and used to instantiate a PointOfInterest object. The PointOfInterest class and PointsOfInterestResponse classes as shown below, respectively.

```
public class PointOfInterest implements Serializable {
    private String name;
    private String info;
    private BigDecimal cost;
    private String formattedCost;


    }


    public class PointsOfInterestResponse implements Serializable {
    private List<PointOfInterest> pointsOfInterest;
    private String totalCostOfTrip;


    public BigDecimal getTotalCost() {
    return pointsOfInterest
    .stream()
    .map(PointOfInterest::getCost)
    .reduce(BigDecimal.ZERO, BigDecimal::add);
    }


    private String error;


    }
```

## Recap

This quick guide looked at incorporating OpenAI's GPT service into a typical Jakarta EE application. This guide uses and recommends the OpenAi-Java library for interacting with the GPT model. It has an intuitive API and builders that make it easy to use. You saw how the Jakarta CDI API construct was incorporated to simplify the creation of the root OpenAiService object. You also saw the use of the MicroProfile Config API to read configuration properties that are used to customise the OpenAiService. Finally, you saw the use of the Jakarta JSON-P API into transforming JSON data to concrete Java types.

AI has gotten the attention of everyone these days after the release of ChatGPT in November of 2022. As an extensible development platform, the Jakarta EE Platform does not tie you down to any specific way to use external services. As you have seen in this brief guide, you can use third party libraries to incorporate AI into your Jakarta EE applications easily using the existing constructs.

As stated in the beginning of this guide, the app from the sample code is deployed and running live on Payara Cloud, the new, automatic Jakarta EE application deployment platform. You can checkout the sample code, play around with it and deploy it to Payara Cloud by simply signing up for a free trial account. Jakarta EE and Payara Cloud together give you everything you need to create the next generation of smart applications using Java.

**sales@payara.fish**

**UK: +44 800 538 5490**
**Intl: +1 888 239 8941**

**www.payara.fish**