Uporaba javanske knjižnice za velike jezikovne modele

Andrej Krajnc, Vojko Ambrožič, Bojan Štok

IZUM – Institut informacijskih znanosti, Maribor, Slovenija
andrej.krajnc@izum.si, vojko.ambrozic@izum.si, bojan.stok@izum.si

Veliki jezikovni modeli omogočajo, da običajnim aplikacijam dodamo funkcionalnosti umetne inteligence. Pametni asistenti z uporabo orodij omogočajo, da veliki jezikovni model preko povratnega klica pokliče poslovno metodo v aplikaciji. V javanskem okolju je najbolj priljubljena knjižnica Langchain4j. S pomočjo Langchain4j lahko uporabimo različne implementacije velikih jezikovnih modelov (OpenAI, Ollama, JLama ...), ne da bi nam bilo treba spreminjati programsko kodo. V prispevku bomo podrobneje predstavili knjižnico Langchain4j in Quarkusovo razširitev Quarkus Langchain4j, ki uporabo knjižnice Langchain4j še bolj poenostavi, saj omogoča deklarativno uporabo knjižnice večinoma le z uporabo anotacij. Tako lahko hitro razvijamo nove funkcionalnosti. Podobni pristop uporablja Spring AI v aplikacijah SpringBoot. Pametnega asistenta COBISS, ki smo ga predstavili lani, smo nadgradili z uporabo knjižnice Langchain4j. Tako je precej ponavljajoče se kode odpadlo. Na kratko bomo predstavili tudi ogrodje Quarkus, ki omogoča zelo hiter zagon aplikacij in je konkurenčno ogrodjema Spring in Jakarta EE.

Ključne besede:
COBISS,
COBISS Lib,
Langchain4j,
Quarkus,
veliki jezikovni modeli.

Univerzitetna založba Univerze v Mariboru
ISBN 978-961-299-037-4
DOI https://doi.org/10.18690/um.feri.7.2025.16

1 Uvod

V zadnjem času se v aplikacije vedno bolj vgrajujejo funkcionalnosti, ki s pomočjo uporabe velikih jezikovnih modelov običajnim aplikacijam dodajo zmogljivosti umetne inteligence. Učenje velikih jezikovnih modelov poteka na osnovi dostopnih podatkov na spletu, zato so dobri v generiranju odgovorov na splošna vprašanja, vendar pa modeli pogosto nimajo domensko specifičnih znanj in zato včasih niso najbolj priročni v primerih, kjer so domensko specifična znanja zelo pomembna. Rešitev za takšne situacije je lahko implementacija lastnega pametnega asistenta.

Na konferenci OTS 2024 smo predstavili izgradnjo pametnega asistenta COBISS, ki smo ga implementirali v okviru aplikacije COBISS Lib za knjižničarje [1]. COBISS Lib podpira postopke nabave monografskih in serijskih publikacij ter elektronskih virov, obdelavo podatkov o zalogi, izvajanje postopkov v izposoji in medknjižnični izposoji ipd.

Ker sta aplikacija in dokumentacija obsežni, smo se odločili razviti asistenta, ki bi knjižničarjem pomagal pri vsakodnevnem delu. Uporabili smo pristop vključevanja agentov in zunanjih virov znanja RAG (ang. Retrieval-Augmented Generation), za jezikovni model pa smo uporabili Chat GPT-4o, čeprav bi lahko uporabili tudi katerikoli drugi model.

Razvili smo programe v Javi 21, ki so markdown-datoteke priročnikov razbile po odstavkih. Za vsak odstavek smo od ChatGPT zahtevali, da pripravi povzetek. Nato smo za vsak tak povzetek pripravili vektor (ang. embeddings) ter ga shranili v vektorsko bazo, ki smo jo kreirali z uporabo javanske knjižnice JVector. Ko uporabnik postavi vprašanje, od ChatGPT zahtevamo, da generira vektor in zazna jezik uporabnika. V vektorski bazi poiščemo pet (nastavitev) najbližjih vektorjev/besedil (ang. cosine similarity) glede na iskalni vektor. Nato od ChatGPT zahtevamo, da za uporabnikovo zahtevo na osnovi najdenih besedil generira odgovor v zaznanem jeziku. Razvili smo javanski odjemalec do ChatGPT z uporabo odprtokodne knjižnice OpenAI-Java.

Celotno pot smo želeli prehoditi sami in pri tem dobiti lastne izkušnje. V prispevku za konferenco OTS 2024 smo najavili, da bomo v prihodnje uporabili najnovejše pristope, s pomočjo katerih lahko veliko hitreje in enostavneje vgradimo funkcionalnosti umetne inteligence. Odločili smo se za uporabo ogrodja Langchain4j, ki je eno od najpopularnejših tovrstnih ogrodij za programski jezik Java.

2 Ogrodje Langchain4j

Langchain4j [2] je odprtokodna javanska knjižnica, ki omogoča enostavno integracijo velikih jezikovnih modelov LLM (ang. Large Language Model) v aplikacije, napisane v Javi. Ustvarjena je bila leta 2023, ko se je pojavila potreba po javanski različici priljubljene knjižnice LangChain, ki je bila napisana v Pythonu. Knjižnica je prilagojena javanskemu ekosistemu in uporablja poznane koncepte, kot so razredi, vmesniki in anotacije. Zasnovana je za razvijalce, ki želijo vgraditi umetno inteligenco v svoje javanske aplikacije. Zadnja verzija Langchain4j 1.2.0 je bila izdana julija 2025.

Knjižnica Langchain4j omogoča, da lahko uporabimo različne implementacije velikih jezikovnih modelov (Ollama, OpenAI, Azure OpenAI, Google Vertex AI Gemini, Jlama, MistralAI …) in različne implementacije vgrajenih shramb (ang. embeddings stores), npr. Cassandra, PGVector (PostgreSQL), Elasticsearch, Redis, Infinispan, MongoDB Atlas, Neo4j itd. Na relativno enostaven način lahko v aplikaciji testiramo različne modele ali shrambe, saj nam ni treba spreminjati aplikacije, ampak spremenimo le konfiguracijo.

Knjižnica Langchain4j podpira tako nizkonivojska orodja, kot so oblikovanje iztočnic, pogovor s pomnilnikom, razčlenjevanje izhodov, kot tudi visokonivojska orodja, kot so storitve AI in RAG.

Langchain4j sam poskrbi za komunikacijo z jezikovnimi modeli in upravljanje konteksta pogovora. Prav tako omogoča vodenje pogovornega spomina (ang. conversation memory), s čimer si lahko aplikacije zapomnijo, kaj je bilo povedano v prejšnjih sporočilih, kar je ključno za naravno interakcijo.

Langchain4j se dobro integrira v sodobna javanska ogrodja, kot so Quarkus, Spring Boot, Helidon in Micronaut. Pomemben mejnik se je zgodil, ko je Red Hat vključil Langchain4j kot uradno razširitev v Quarkus, kar je prineslo še večjo priljubljenost Langchain4j v javanskem ekosistemu.

2.1. Uporaba Langchain4j v javanskih strežniških aplikacijah

Knjižnico Langchain4j lahko uporabljamo v javanskih aplikacijah na različne načine.

V klasičnih javanskih aplikacijah, kjer se uporablja standardna verzija Jave (ang. Java Standard Edition), lahko Langchain4j uporabljamo na preprost način kot navadno knjižnico razredov. Takšen pristop je primeren za enostavne primere, skripte, CLI-orodja, testiranje ipd.

Primer uporabe Langchain4j kot knjižnice razredov:

```
ChatModel chatModel = OpenAiChatModel.builder().apiKey(OPENAI_API_KEY).build();
String answer = chatModel.chat("Kaj je Langchain4j?");
```

V javanskih strežniških aplikacijah (ang. Enterprise Java) lahko uporabljamo Langchain4j na bolj napreden način, saj lahko uporabimo različne stvari, kot so anotacije, vrivanje odvisnosti (ang. dependency injection) in druge tehnologije, ki so značilne za javanske strežniške aplikacije. Takšen pristop je primeren za izgradnjo vmesnikov REST, mikrostoritev in raznih spletnih aplikacij.

Langchain4j lahko uporabljamo v ogrodjih, kot so Spring, Quarkus, Micronaut, Helidon ipd. Kar nekaj ogrodij je integriralo Langchain4j na način, ki še olajša uporabo umetne inteligence v javanskih strežniških aplikacijah. Tako obstajajo integracije, kot so Langchain4j Spring, Quarkus Langchain4j, Langchain4j Microprofile (SmallRye LLM) itd. Te integracije omogočajo, da lahko umetno inteligenco v aplikacijah uporabimo z zgolj nekaj vrsticami kode.

Primer kode v Quarkus Langchain4j:

```
@ApplicationScoped
public class LlmService {
   @Inject
   OpenAiChatModel chatModel;

   public String answer(String question) {
      return chatModel.chat(question);
   }
}
```

2.2. Quarkus Langchain4j

Čeprav večina naših aplikacij temelji na uporabi Java EE oz. Jakarta EE, smo se pri implementaciji pametnega asistenta COBISS odločili za uporabo Quarkus Langchain4j [3].

Quarkus je sodobno javansko ogrodje, zasnovano za hitro in učinkovito izvajanje v oblačnih okoljih, npr. v Kubernetes, OpenShift, Docker Swarm in podobno [4]. Prva verzija 1.0 je bila narejena znotraj podjetja Red Hat leta 2019, zadnja LTS verzija 3.20.2 je iz julija 2025. Glavni cilj ogrodja Quarkus je optimizirati zagon in delovanje aplikacij, zato ga odlikujeta izjemno hiter zagon in majhna poraba pomnilnika. Ogrodje ima zelo dobro podporo za navidezni stroj **GraalVM**, s pomočjo katerega lahko ustvarjamo izvorne slike (ang. native image). Izvorne slike so majhne binarne datoteke, ki se zaženejo zelo hitro in so med izvajanjem neodvisne od platforme Java.

Quarkus je celovito strežniško ogrodje (ang. full stack server framework), ki ne vključuje uporabniškega vmesnika (HTML/JS/CSS), ampak ponuja vse ključne tehnologije za razvoj celotne strežniške aplikacije, tj. od podatkovnega sloja do programskih vmesnikov, varnosti in povezav z zunanjimi sistemi. Quarkus temelji na veliko strežniških standardih, pri razvoju katerih je že leta sodelovalo podjetje Red Hat. Tako ogrodje podpira vrivanje odvisnosti preko standarda CDI (ang. Contexts & Dependency Injection), izdelavo vmesnikov REST preko standarda JAX-RS, dostop do podatkovnih baz preko standarda JPA (Java Persistence API), omogoča uporabo transakcijskega programskega vmesnika JTA (Java Transaction API), itd. Quarkus je združljiv z javanskimi standardi, a hkrati prinaša inovacije, ki jih v drugih ogrodjih pogosto ni.

Quarkus Langchain4j je razširitev, ki prinaša zmogljivosti umetne inteligence v okolje Quarkus na način, skladen z njegovimi načeli: hiter zagon, majhna poraba pomnilnika in oblačni razvoj. Razširitev omogoča, da razvijalci uporabljajo jezikovne modele LLM (ang. Large Language Model) znotraj aplikacij Quarkus z minimalno konfiguracijo, izkoriščajo pa lahko vse funkcionalnosti Quarkusa za razvoj in integracijo z drugimi storitvami.

Ker je Langchain4j uradni del Quarkusa, je z njo mogoče brez težav povezati jezikovne modele LLM z obstoječo infrastrukturo (npr. podatkovne baze, sporočilni sistemi, varnostni mehanizmi). Deluje brez težav z izvornimi slikami (GraalVM), kar pomeni, da lahko zgradimo mikrostoritve, temelječe na umetni inteligenci, ki so lahke, hitro odzivne in pripravljene za izvajanje v oblačnih okoljih, kot Kubernetes, OpenShift, Docker Swarm itd. Poleg tega omogoča enostavno vključevanje agentov in zunanjih virov znanja (RAG) v tipičen Quarkusov razvojni cikel.

Primer implementacije enostavnega klepeta:

```
@ApplicationScoped
@RegisterAiService()
public interface DemoChat {

  String chat(@MemoryId int id, @UserMessage String message);

  @SystemMessage("you are java programmer")
  String chatJava(@MemoryId int id, @UserMessage String message);
}
```

Z anotacijo @SystemMessage lahko damo dodatna navodila sistemu, z anotacijo @MemoryId določamo ID klepeta, z anotacijo @UserMessage posredujemo sporočilo uporabnika.

Če uporabimo multimodalen model jezik-slika (ang. multimodal language-image), lahko razpoznavamo vsebino slike ali slike generiramo. Eden izmed odprtokodnih modelov je LLaVa [5].

Primer kode za razpoznavanje slike:

```
@ApplicationScoped
@RegisterAiService()
public interface DemoImageService {

  @UserMessage(value = "What do you see")
  String describeImage(@ImageUrl String imageUrl);

  @UserMessage(value = "Extract only text from image")
  String extractText(@ImageUrl String imageUrl);
}
```

Nekateri modeli omogočajo tudi generiranje slike. Primer kode:

```
@ApplicationScoped
@RegisterAiService()
```

```
public interface DemoImageService {
  Image generateImage(String message);
}
```

3 Implementacija pametnega asistenta COBISS z uporabo Langchain4j

Prvo verzijo pametnega asistenta COBISS smo razvili z uporabo ogrodja **Jakarta EE** na aplikacijskem strežniku **WildFly**. Asistenta smo nadgradili tako, da smo namesto ogrodja Jakarta EE uporabili Quarkus Langchain4j. Komponente za dostop do velikih jezikovnih modelov (OpenAI-Java) smo zamenjali s knjižnico **Langchain4j**. Prav tako smo vektorsko bazo **jVector** nadomestili z bazo **PGVector** (na osnovi PostgreSQL).

Aplikacijo smo testirali z uporabo različnih modelov **OpenAI** in z uporabo odprtokodnih modelov, kot so »llama3.3«, »deepseek«, ki so bili nameščeni na našem lokalnem strežniku in dostopni preko vmesnika **Ollama**.

Največja sprememba je bila na področju priprave in obdelave dokumentov, ki se nato shranijo v vektorsko bazo in uporabljajo pri iskanju konteksta v okviru strategije **RAG** (ang. Retrieval-Augmented Generation**)**.

3.1. Inicializacija modelov in podatkovnih shramb

Običajno se ob dvigu aplikacije ali prvi uporabi pametnega asistenta na osnovi konfiguracije inicializirajo različni modeli in podatkovne shrambe.

V primeru pametnega asistenta COBISS uporabljamo model za pogovore (ang. chat model) ter model za vdelave (ang. embedding model). Primer kode za inicalizacijo modela za pogovore:

```
ChatModel chatModel = OpenAiChatModel.builder().
  .apiKey(OPENAI_API_KEY)
  .modelName(“gpt-4o”)
  .temperature(0)
  .maxTokens(3000)
  .build();


 ChatModel chatModel = OllamaLanguageModel.builder().
  .baseUrl(OLLAMA_URL)
  .modelName("llama3.3")
  .temperature(0)
  .maxTokens(3000)
  .build();
```

Primer kode za inicalizacijo modela za vdelave:

```
EmbeddingModel embeddingModel =  OpenAiEmbeddingModel.builder()
  .apiKey(OPENAI_API_KEY)
  .modelName("text-embedding-3-small")
  .build();


EmbeddingModel embeddingModel =  OllamaEmbeddingModel.builder()
  .baseUrl(OLLAMA_URL)
  .modelName("llama3.3")
  .build();
```

Primer kode za inicializacijo podatkovne shrambe vdelav:

```
EmbeddingStore<TextSegment> embeddingStore = PgVectorEmbeddingStore.builder()
  .host(PGVECTOR_HOST)
  .port(PGVECTOR_PORT)
  .createTable(recreateTable)
  .dropTableFirst(recreateTable)
  .dimension(MODEL_DIMENSION)
  .table(TABLE_NAME)
  .user(USERNAME)
  .password(PASWORD)
  .database(DATABASE)
  .build();
```

3.2. Priprava poslovnih podatkov

Pri uporabi strategije RAG je ključnega pomena ustrezna priprava poslovnih podatkov za hrambo v vektorski bazi. Knjižnica Langchain4j daje podporo za branje dokumentov iz različnih virov, kot so lokalni diski ali spletni naslovi. Podprti formati vključujejo HTML, PDF, TXT in druge. V našem primeru je bila vsebina priročnikov v **HTML**-obliki.

Prvi korak: Razčlenjevanje dokumentov

Dokumente je treba najprej razčleniti, torej pretvoriti njihovo vsebino v besedilno obliko, ki jo lahko obdelujemo. Langchain4j omogoča uporabo različnih razčlenjevalnikov (ang. parsers) za ta namen.

V prvem koraku poiščemo relevantne HTML-dokumente:

```
try (Stream<Path> stream = Files.walk(Paths.get(documentsPath))) {
  stream.filter(Files::isRegularFile).filter(path -> {
    String fileName = path.getFileName().toString().toLowerCase();
    return fileName.endsWith(".html");
  }).forEach(path -> parseDocument(path);
} catch (IOException e) {
  e.printStackTrace();
}
```

Nato preberemo vsak najdeni dokument in mu po potrebi dodamo več metapodatkov:

```
DocumentParser parser = new ApacheTikaDocumentParser();
Document document = FileSystemDocumentLoader.loadDocument(path, parser);
document.metadata().put(Document.URL, documentUrl);
document.metadata().put("document_type", »LOAN« );
```

Drugi korak: Preoblikovanje vsebine (opcijsko)

Po razčlenjevanju lahko dokumente dodatno preoblikujemo: odstranimo nepotrebne dele, izboljšamo formatiranje ali dodamo metapodatke. Namen tega koraka je izboljšati kakovost informacij, ki bodo kasneje predstavljene jezikovnemu modelu.

```
DocumentTransformer transformer = new DocumentTransformer(
  public Document transform(Document document) {
    String text = document.text();
    text = format(text);
    document = new Document(text.trim(), document.metadata());
```

```
    return document;
  }
);
Document transformedDocument = transformer.transform(document);
```

Tretji korak: Razdelitev vsebine na segmente

Eden ključnih korakov je razdelitev dokumenta na manjše, bolj obvladljive enote (ang. chunks). Veliki jezikovni modeli imajo omejeno velikost vhodnega okna (ang. context window), zato mora biti vsak segment dovolj majhen, da ga model lahko obdela v enem pozivu.

Langchain4j ponuja razrede, ki omogočajo razdelitev vsebine glede na število besed, stavkov ali znakov, in po potrebi omogočajo prekrivanje med segmenti za boljši kontekst.

```
DocumentSplitter splitter = DocumentSplitters.recursive(5000, 100);
List<TextSegment> segments = splitter.split(document);
for (TextSegment segment : segments) {
  saveEmbedding(segment, documentType);
}
```

Četrti korak: Izračun vektorjev in shramba v vektorsko bazo

Za vse kreirane segmente izračunamo numerično reprezentacijo vsebine ter izračunane vektorje shranimo v vektorsko bazo:

```
Embedding embedding = embeddingModel.embed(segment).content();
embeddingStore.add(embedding, segment);
```

Shranjena vrstica vsebuje tekst, metapodatke in izračun večdimenzionalnega vektorja segmenta.

3.3. Prikaz poslovnih podatkov

RAG (ang. Retrieval-Augmented Generation) je metoda, pri kateri veliki jezikovni model (LLM) pri generiranju odgovorov uporablja zunanje podatkovne vire. Namesto da bi se zanašal le na lastno znanje, sistem najprej poišče relevantne informacije v vnaprej pripravljeni vektorski bazi, nato pa te vsebine vključi v poziv (ang. prompt) v modelu. Tako nastane odgovor, ki je bolj dejstveno utemeljen, aktualen in prilagojen konkretni podatkovni zbirki.

RAG združuje prednosti iskanja po vsebini in naravnega jezika za natančnejše, zanesljivejše odgovore.

Iztočnica uporabnika

Uporabnik zastavi vprašanje ali poda zahtevo v obliki besedilnega vnosa (t.i. prompt).

Iskanje po vektorski bazi

Sistem pretvori vprašanje v vektorsko predstavitev ter poišče najbolj relevantne dokumente ali odlomke iz vektorske baze podatkov (npr. PGVector), ki so semantično podobni vprašanju.

Primer kode za iskanje:

```
Embedding embeddedQuestion = embeddingModel.embed(question).content();
EmbeddingStore<TextSegment> embStore = getEmbeddingStore();
EmbeddingSearchRequest searchRequest = EmbeddingSearchRequest.builder()
  .queryEmbedding(embeddedQuestion)
  .maxResults(5)
  .minScore(0.8)
```

```
  .build();
EmbeddingSearchResult<TextSegment> searchResult = embStore.search(searchRequest);
```

Priprava odgovora

Najdeni odlomki se dodajo kot kontekst jezikovnemu modelu, ki na tej osnovi generira relevanten odgovor, podprt z dejstvi.

```
ChatMemory chatMemory = MessageWindowChatMemory.withMaxMessages(20);
SystemMessage systemMsg = new SystemMessage("""
Use these guidelance to help the user:
    - You are a librarian
    - When listing, use list notation
    - Answer the user prompt with help of supplied text
    """);
chatMemory.add(systemMsg);
searchResult.matches().forEach(match -> {
     chatMemory.add(UserMessage.from(match.embedded().text()));
});
UserMessage userQuestion = UserMessage.from(question);
chatMemory.add(userQuestion);
AiMessage aiAnswer = chatModel.chat(chatMemory.messages()).aiMessage();
chatMemory.add(aiAnswer);
```

Če primerjamo odgovore, ki nam jih dajeta OpenAI in Ollama, lahko rečemo, da OpenAI daje boljše odgovore in to v krajšem času, vendar pa je uporaba orodja Ollama na lastnem strežniku lahko cenejša rešitev, ki zagotavlja tudi zasebnost, saj vsi podatki ostajajo v lokalnem okolju.

Priprava odgovora v obliki toka

Včasih ne želimo čakati na dokončno oblikovan odgovor, temveč želimo odgovor dobivati sproti, kot se pripravlja. Tako prihaja odgovor v koščkih v obliki toka, kot smo ga vajeni iz orodij, kot je na primer ChatGPT. Na ta način lahko zagotovimo hitrejši odziv na uporabniškem vmesniku, kar predstavlja boljšo uporabniško izkušnjo.

Če želimo dobiti odgovore na takšen način, moramo uporabljati drug model za pogovore, in sicer model v obliki toka (ang. streaming model).

Primer kode za inicializacijo modela za pogovore v obliki toka:

```
StreamingChatModel streamingChatModel= OpenAiStreamingChatModel.builder().
  .apiKey(OPENAI_API_KEY)
  .modelName(“gpt-4o”)
  .temperature(0)
  .maxTokens(3000)
  .build();


 StreamingChatModel streamingChatModel = OllamaStreamingChatModel.builder().
  .baseUrl(OLLAMA_URL)
  .modelName("llama3.3")
  .temperature(0)
  .maxTokens(3000)
  .build();
```

V primeru uporabe tokov je tudi koda za pripravo odgovora drugačna:

```
@POST
@Path("/streamHelp")
@Consumes({"application/json"})
public Response streamHelp(AiRequest data) {
  StreamingOutput stream = output -> {
    PrintWriter writer = new PrintWriter(new OutputStreamWriter(output, StandardCharsets.UTF_8));
    aiService.getStreamHelp(data, writer);
  };
  return Response.ok(stream).build();
}

CountDownLatch latch = new CountDownLatch(1);
streamingChatModel.chat(chatMemory.messages(), new StreamingChatResponseHandler() {
  @Override
  public void onPartialResponse(String partialResponse) {
    writer.write(partialResponse);
    writer.flush();
  }
  @Override
  public void onCompleteResponse(ChatResponse completeResponse) {
    writer.println("\n$DOCUMENTS\n");
    for (AiDocument document : documents) {
      writer.write("Document: " + document.getDocumentUrl() + "\n");
    }
    writer.flush();
    latch.countDown();
  }
  @Override
  public void onError(Throwable error) {
    writer.write("Error: " + error.getMessage());
    writer.flush();
    latch.countDown();
  }
});
latch.await();
```

4 Orodja Langchain4j

V Langchain4j je vgrajen koncept orodij (ang. tools) in klicanja funkcij (ang. function calling).

Orodja LangChain4j [6] so zunanje funkcije ali storitve, ki jih jezikovni model lahko po potrebi uporabi kot pomoč. To modelu omogoča, da ne odgovarja na vprašanja zgolj na klasičen način, temveč odgovore obogati, tako da pokliče zunanje funkcije, kot so na primer matematični izračuni, iskanje podatkov v bazi, pridobivanje podatkov s spleta ali uporaba programskih vmesnikov (API).

Novo orodje (ang. tool) lahko ustvarimo na različne načine. Najbolj enostaven način je, da implementiramo običajno javansko metodo in jo označimo z anotacijo @Tool. Ko model sklepa, da uporabnik želi nekaj, kar orodje zna narediti, ga samodejno pokliče.

Primer kode za definiranje novega orodja za seštevanje:

```
public class CobissTool {
  @Tool("Avtor knjige z naslovom")
  int getBookAuthor(String title) {
    List<Book> books = cobissSearch.searchTitle(title);
    if (books.size() > 0) return books.get(0).getAuthor();
    return null;
  }
}
```

Novo orodje registriramo na enostaven način:

```
@RegisterAiService(tools=CobissTool.class)
```

Ko uporabnik pošlje sporočilo, jezikovni model oceni, ali je katero od orodij primerno za pomoč pri generiranju odgovora in v primeru, da uporabnik želi pridobiti avtorja neke knjige, uporabi orodje za pridobivanje avtorja in na koncu odgovori.

```
String answer = assistant.chat("Kdo je avtor knjige z naslovom Na klancu?");
```

Primer odgovora na koncu je: "Avtor knjige z naslovom Na klancu je Ivan Cankar.".

Orodij Langchain4j zaenkrat še nismo uporabili v pametnem asistentu COBISS, jih pa nameravamo v prihodnje.

5 Zaključek

V prispevku smo prikazali implementacijo pametnega asistenta COBISS z uporabo javanske knjižnice Langchain4j. Langchain4j je v tem trenutku zagotovo eno najbolj vročih orodij za izgradnjo javanskih aplikacij, ki uporabljajo zmožnosti jezikovnih modelov.

Glede na prvotni pristop, kjer smo vse preizkusili ročno, nam je Langchain4j zelo olajšal izgradnjo pametnega asistenta COBISS, saj ponuja številne uporabne funkcionalnosti. Orodje podpira najrazličnejše implementacije jezikovnih modelov, preklopi med njimi pa so enostavni, saj običajno za uporabo drugega modela ni treba spreminjati programske kode, temveč le konfiguracijo.

Zaenkrat smo za implementacijo pametnega asistenta COBISS uporabili osnovne funkcionalnosti Langchain4j, v prihodnje pa nameravamo uporabiti tudi orodja Langchain4j na način, da bodo jezikovni modeli klicali naše funkcije.

Preizkusiti nameravamo tudi uporabo vnaprej pripravljenih okolij Docker za lokalno poganjanje jezikovnih modelov (Docker Model Runner) [7]. Ker se modeli tako poganjajo lokalno, ni potrebe po klicanju zunanjih programskih vmesnikov, s čimer zmanjšamo stroške in ohranjamo zasebnost pri uporabi jezikovnih modelov.

Literatura

[1] AMBROŽIČ, Vojko, KRAJNC, Andrej, ŠTOK, Bojan. Primer razvoja pametnega asistenta. V: PAVLIČ, Luka (ur.), BERANIČ, Tina (ur.), HERIČKO, Marjan (ur.). OTS 2024 : sodobne informacijske tehnologije in storitve : zbornik 27. konference : [Maribor, 4. in 5. september 2024]. 1. izd. Maribor: Univerza v Mariboru, Univerzitetna založba, 2024. Str. 15-24, ilustr. ISBN 978-961-286-893-2. https://press.um.si/index.php/ump/catalog/book/903/chapter/68, DOI: 10.18690/um.feri.4.2024.2

[2] LangChain4j, https://github.com/langchain4j/langchain4j/, obiskano 29. 7. 2025

[3] Quarkus LangChain4j, https://docs.quarkiverse.io/quarkus-langchain4j/dev/index.html/, obiskano 29. 7. 2025

[4] Quarkus, https://quarkus.io/, obiskano 29. 7. 2025

[5] LLaVa, https://llava-vl.github.io/, obiskano 29. 7. 2025

[6] LangChain4j Tools (Function Calling), https://docs.langchain4j.dev/tutorials/tools, obiskano 29. 7. 2025

[7] Docker Model Runner, https://docs.docker.com/ai/model-runner/, obiskano 29. 7. 2025