

Session 5: Building AI-Powered Java Apps

Spring AI, RAG, Tools, and MCP

Press Space for next page →



Welcome to Session 5!

Building AI-Powered Java Applications

From Spring Boot to Spring AI

Spring Boot 3.5 + Spring AI 1.1.0 + Java 21

What We'll Build Today

- **Spring AI ChatClient** - Fluent API for LLM interactions
- **Prompt Templates** - Reusable, parameterized prompts
- **RAG Pipeline** - Chat with your documents
- **Function Calling** - Give AI tools to execute code
- **MCP Integration** - Enhanced context for Cursor

Course Journey

- **Session 1:** Cursor fundamentals
- **Session 2:** Mobile development with AI
- **Session 3:** Agentic coding patterns
- **Session 4:** AI-assisted testing
- **Session 5:** Building AI apps with Spring AI ← Today

Today's Stack

Spring Boot 3.5.7 • Spring AI 1.1.0 • Java 21

- OpenAI or Anthropic API keys required
- All code available in `spring-ai-demo/` folder
- Labs guide you through each feature

Part 1: Introduction to Spring AI

The Spring Way to Build AI Applications

- Official Spring project for AI integration
- Portable abstraction over AI providers
- Familiar Spring programming model



What is Spring AI?

- **Official Spring Project** for AI integration
- **Portable abstraction** over AI providers
- **Spring Boot auto-configuration**
- **Familiar Spring programming model**

Spring AI Core Components

- **ChatClient:** Fluent API for LLM interactions
- **Embeddings:** Vector representations of text
- **Vector Stores:** Storage for document embeddings
- **Function Calling:** Tools that AI can invoke
- **Document Readers:** PDF, Word, text processing

Spring AI Advantages

- Switch between OpenAI, Anthropic, Ollama without code changes
- Dependency injection for AI components
- Spring Boot conventions and auto-configuration
- Familiar patterns: RestTemplate → ChatClient

Spring AI Gradle Dependencies

```
1  plugins {
2      id 'java'
3      id 'org.springframework.boot' version '3.5.7'
4      id 'io.spring.dependency-management' version '1.1.7'
5  }
6
7  ext {
8      set('springAiVersion', "1.1.0")
9  }
10
11 dependencies {
12     implementation 'org.springframework.ai:spring-ai-starter-model-openai'
13 }
14
15 dependencyManagement {
16     imports {
17         mavenBom "org.springframework.ai:spring-ai-bom:${springAiVersion}"
18     }
19 }
```

Spring AI Configuration

```
1 # application.properties  
2 spring.ai.openai.api-key=${OPENAI_API_KEY}  
3 spring.ai.openai.chat.options.model=gpt-5-nano  
4 spring.ai.openai.chat.options.temperature=1.0
```

- Set `OPENAI_API_KEY` environment variable
- Or use `.env` file with Spring Boot
- Alternative: Use Anthropic with `spring-ai-anthropic`

```
import express from 'express';
import mongoose from 'mongoose';
const User = mongoose.model('User');

// GET register
router.get('/register', function(req, res, next) {
  return res.render('register', { title: 'Sign Up' });
});

// POST /register
router.post('/register', function(req, res, next) {
  if (!req.body.email || !req.body.name || !req.body.favoriteBook || !req.body.password || !req.body.confirmPassword) {
    // confirm that user typed same password twice
    if (req.body.password !== req.body.confirmPassword) {
      var err = new Error('Passwords do not match.');
      err.status = 400;
      return next(err);
    }
  }

  // create user object with form input
  var userData = {
    email: req.body.email,
    name: req.body.name,
    favoriteBook: req.body.favoriteBook,
    password: req.body.password
  };

  // use schema's 'create' method to insert document into Range
  User.create(userData, function(error, user) {
    if (error) {
      return next(error);
    }
  });
});

app.listen(3001, () => console.log('Authenticating...'));

```

Part 2: Chat Client & Templating

Fluent API for LLM Interactions

- Build requests with fluent builder pattern
- System and user message configuration
- Prompt templates for reusable prompts

ChatClient Basics

```
1  @RestController
2  @RequestMapping("/api/chat")
3  public class ChatController {
4
5      private final ChatClient chatClient;
6
7      public ChatController(ChatClient.Builder builder) {
8          this.chatClient = builder.build();
9      }
10
11     @GetMapping
12     public String chat(@RequestParam String message) {
13         return chatClient.prompt()
14             .user(message)
15             .call()
16             .content();
17     }
18 }
```

ChatClient Features

- **Fluent API** for building requests
- **System and user messages** configuration
- **Response parsing** and handling
- **Streaming responses** (optional)

Agent Mode Prompt:

- 1 Create a ChatController with a GET endpoint /chat.
- 2 Inject ChatClient.Builder and return LLM response.

System Prompts

```
1  @GetMapping("/expert")
2  public String expertChat(@RequestParam String topic) {
3      return chatClient.prompt()
4          .system("""
5              You are an expert software architect
6              specializing in Spring Boot applications.
7              Provide concise, practical advice.
8              """)
9      .user("How do I implement " + topic)
10     .call()
11     .content();
12 }
```

Structured Responses

```
1 record BookReview(String title, int rating, String summary) {}
2
3 @GetMapping("/review")
4 public BookReview getBookReview(@RequestParam String book) {
5     return chatClient.prompt()
6         .user("Write a review of the book: " + book)
7         .call()
8         .entity(BookReview.class);
9 }
```

Spring AI automatically: Generates JSON schema → Instructs LLM → Parses to Java object

Prompt Templates

Template File: src/main/resources/prompts/joke.st

- ```
1 Tell me a {style} joke about {topic}.
2 Make it appropriate for a professional audience.
```

## Key Points:

- StringTemplate format (.st files)
- Variable substitution with {variableName}
- Version control your prompts

# Using Prompt Templates

```
1 @GetMapping("/joke")
2 public String tellJoke(
3 @RequestParam String topic,
4 @RequestParam(defaultValue = "funny") String style) {
5
6 return chatClient.prompt()
7 .user(u -> u.text(
8 "classpath:/prompts/joke.st",
9 Map.of("topic", topic, "style", style)
10))
11 .call()
12 .content();
13 }
```

# Student Exercise: Chat Interface

Time: 10 minutes

1. **Create** a new controller
2. **Inject** ChatClient.Builder
3. **Add** a system prompt for your domain
4. **Create** a template for common queries
5. **Test** with various inputs

# Challenge Ideas

- **Code reviewer** - Analyze Java code snippets
- **Documentation generator** - Create JavaDoc
- **SQL translator** - Natural language to SQL
- **Tech explainer** - Simplify complex topics

**Bonus:** Use `.entity()` to return structured responses as Java records

# Part 3: Retrieval Augmented Generation (RAG)

## Chat with Your Documents

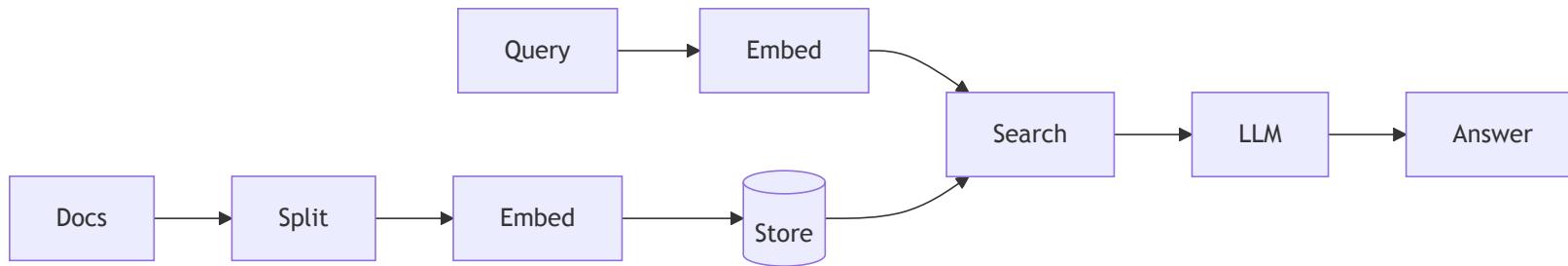
- Ground AI responses in your data
- Vector stores for semantic search
- Document chunking and embeddings



# Why RAG?

- **Ground AI in your data** - Not just training data
- **Prevent hallucinations** - Provide context
- **Domain-specific knowledge** - Your documents, policies, code
- **Up-to-date information** - Add new docs anytime

# RAG Pipeline



# RAG Key Concepts

- **Chunking:** Split documents into manageable pieces
- **Embeddings:** Convert text to vectors (meaning as numbers)
- **Vector Similarity:** Find chunks semantically similar to query
- **Context Injection:** Add retrieved chunks to LLM prompt

# Document Ingestion

```
1 @Component
2 public class DocumentLoader implements CommandLineRunner {
3 private final VectorStore vectorStore;
4
5 public DocumentLoader(VectorStore vectorStore) {
6 this.vectorStore = vectorStore;
7 }
8
9 @Override
10 public void run(String... args) {
11 Resource resource = new ClassPathResource("documents/policy.txt");
12 TextSplitter splitter = new TokenTextSplitter();
13 List<Document> documents = splitter.split(new TextReader(resource).get());
14 vectorStore.add(documents); // Generate embeddings and store
15 }
16 }
```

- Load document → Split into chunks → Embed → Store vectors

# SimpleVectorStore (In-Memory)

```
1 @Configuration
2 public class VectorStoreConfig {
3
4 @Bean
5 public VectorStore vectorStore(EmbeddingModel embeddingModel) {
6 return SimpleVectorStore.builder(embeddingModel).build();
7 }
8 }
```

**Good for:** Development, testing, small document sets, prototypes

# Production Vector Stores

- **Chroma** - Open source, easy setup
- **Pinecone** - Managed service
- **PgVector** - PostgreSQL extension
- **Redis** - If already using Redis

```
1 @Bean
2 public VectorStore vectorStore(JdbcTemplate jdbc, EmbeddingModel model) {
3 return new PgVectorStore(jdbc, model);
4 }
```

# RAG with QuestionAnswerAdvisor

Spring AI provides the `QuestionAnswerAdvisor` to handle RAG automatically:

```
1 @RestController
2 @RequestMapping("/api/rag")
3 public class RagController {
4
5 private final ChatClient chatClient;
6
7 public RagController(ChatClient.Builder builder, VectorStore vectorStore) {
8 this.chatClient = builder
9 .defaultAdvisors(
10 QuestionAnswerAdvisor.builder(vectorStore)
11 .searchRequest(SearchRequest.builder().topK(5).build())
12 .build())
13 .build();
14 }
}
```

# RAG Query - Simple!

```
1 @GetMapping("/query")
2 public String query(@RequestParam String question) {
3 return chatClient.prompt()
4 .user(question)
5 .call()
6 .content();
7 }
8 }
```

**The Advisor handles:** similarity search, context injection, prompt augmentation

# QuestionAnswerAdvisor Benefits

- **No manual search** - Advisor queries vector store automatically
- **Context injection** - Retrieved documents added to prompt
- **Configurable** - Set topK, similarity threshold, filters
- **Clean code** - Controller stays simple

# Chunking Strategies

- **TokenTextSplitter** - By token count (most common)
- **Paragraph splitter** - Natural boundaries
- **Sliding window** - Overlap for continuity

**Sweet spot:** 300-800 tokens with 10-20% overlap

# Search Configuration

```
1 SearchRequest query(question)
2 .withTopK(5) // Return top 5 matches
3 .withSimilarityThreshold(0.7) // Min similarity
4 .withFilterExpression("type == 'policy'"); // Metadata filter
```

- Explicitly tell AI to use context
- Handle "I don't know" gracefully
- Test with questions not in docs

# Student Exercise: RAG Pipeline

Time: 20 minutes

1. **Create** sample documents in `src/main/resources/documents/`
2. **Configure** VectorStore bean
3. **Implement** DocumentLoader
4. **Create** RAG endpoint
5. **Test** with queries requiring document knowledge

# RAG Challenge Ideas

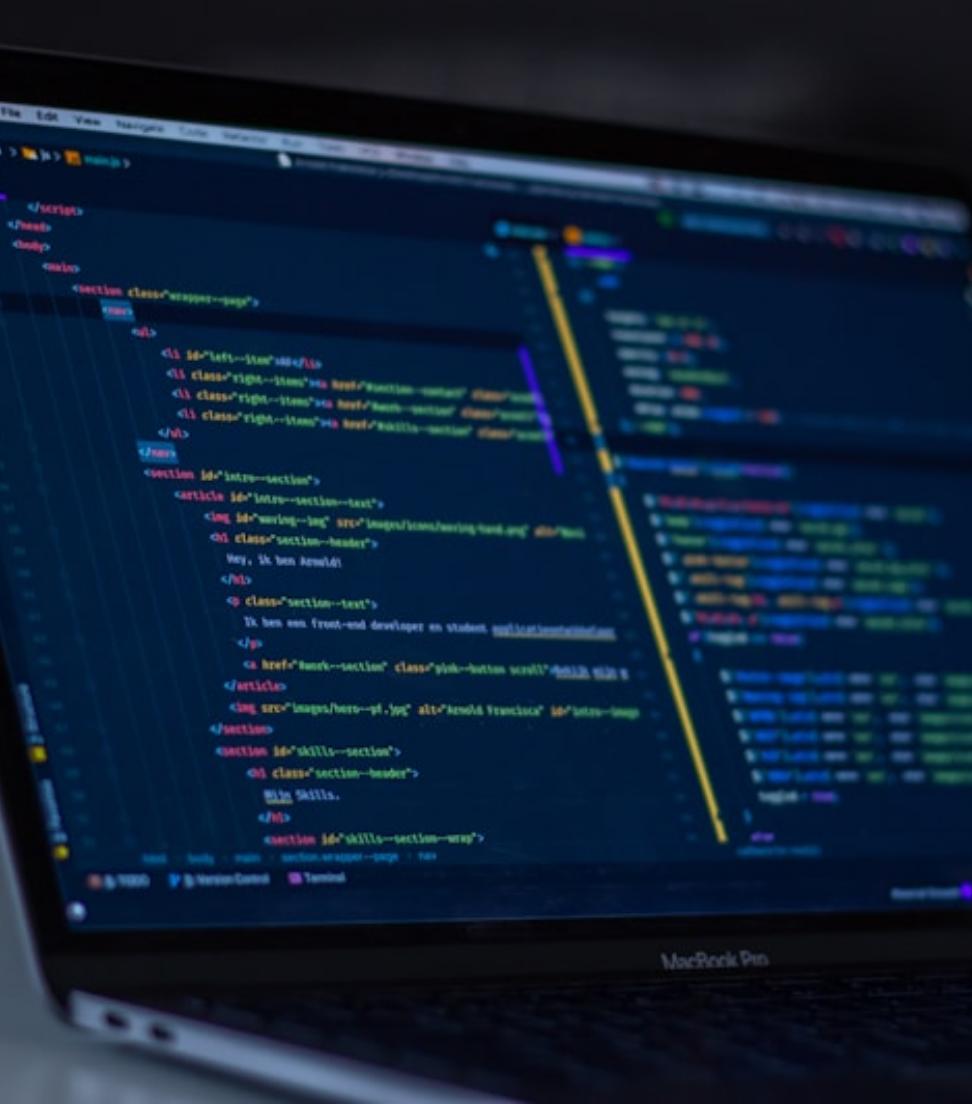
- **Company policies** - HR handbook, procedures
- **Technical docs** - API documentation
- **Knowledge base** - FAQ, troubleshooting

**Bonus:** Add metadata (author, date) and filter search results

# BREAK

10 minutes

Grab coffee before we dive into function calling!



# Part 4: Tools & Function Calling

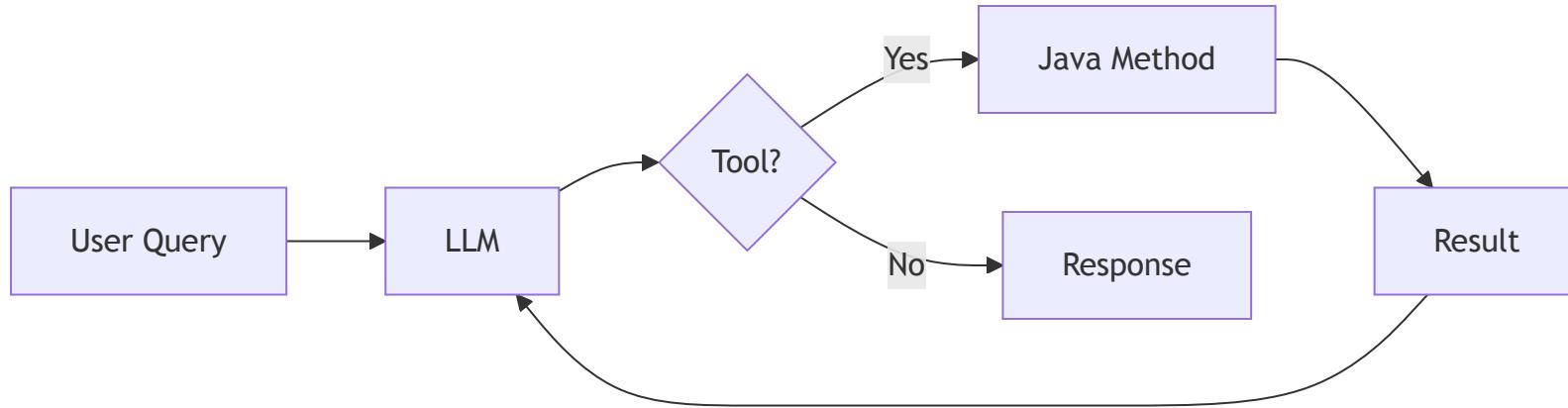
Give AI the Ability to Execute Code

- AI decides when to call functions
- Structured output for function invocation
- Multi-step workflow automation

# What is Function Calling?

- **AI decides** when to call functions
- **Structured output** from LLM (function name + args)
- **Your code executes** the function
- **Return result** to AI for response

# Function Calling Flow



# Function Calling Use Cases

- **Database lookups** - Query data
- **API calls** - External services
- **Calculations** - Math, business logic
- **Workflow automation** - Multi-step tasks

**Key:** AI can call your Java methods based on natural language!

# Defining Tools with @Tool Annotation

```
1 import org.springframework.ai.tool.annotation.Tool;
2 import org.springframework.ai.tool.annotation.ToolParam;
3
4 @Component
5 public class WeatherTools {
6
7 public record WeatherResponse(
8 String location, String temperature, String description) {}
```

- Tools are Spring `@Component` beans
- Define response types as records
- Import `@Tool` and `@ToolParam` annotations

# @Tool Method Definition

```
1 @Tool(description = "Get current weather for a location."
2 + "Use when user asks about weather conditions.")
3 public WeatherResponse getCurrentWeather(
4 @ToolParam(description = "City and state, e.g. 'San Francisco, CA'")
5 String location,
6 @ToolParam(description = "Unit: 'celsius' or 'fahrenheitz'",
7 required = false)
8 String unit) {
9 String temp = "celsius".equalsIgnoreCase(unit) ? "22°C" : "72°F";
10 return new WeatherResponse(location, temp, "Sunny with light clouds");
11 }
12 }
```

# @Tool Annotation Benefits

- **Declarative** - Mark methods directly with `@Tool`
- **Type-safe** - Use primitives, records, lists, maps
- **Self-documenting** - `@ToolParam` describes each parameter
- **Spring integrated** - Tools are just `@Component` beans

# Register Tools with ChatClient

```
1 @RestController
2 @RequestMapping("/api/tools")
3 public class ToolController {
4
5 private final ChatClient chatClient;
6
7 public ToolController(ChatClient.Builder builder, WeatherTools weatherTools) {
8 this.chatClient = builder
9 .defaultTools(weatherTools) // Register all @Tool methods
10 .build();
11 }
12
13 @GetMapping("/chat")
14 public String chat(@RequestParam String message) {
15 return chatClient.prompt().user(message).call().content();
16 }
17 }
```

# Function Calling in Action

User: "What's the weather in San Francisco?"

1. AI recognizes need for `weatherFunction`
2. Extracts location: "San Francisco"
3. Calls your Java function
4. Receives: 72°F, Sunny
5. Responds: "It's currently 72°F and sunny..."

# Database Access with @Tool

```
1 @Component
2 public class CustomerTools {
3
4 public record Customer(Long id, String name, String email, String tier) {}
5
6 private final Map<String, Customer> customers = Map.of(
7 "alice@example.com", new Customer(1L, "Alice Smith", "alice@example.com", "Gold"),
8 "bob@example.com", new Customer(2L, "Bob Jones", "bob@example.com", "Silver")
9);

```

Tools can access any Spring-managed resource: repositories, services, APIs

# CustomerTools @Tool Method

```
1 @Tool(description = "Find a customer by email address")
2 public Customer findCustomerByEmail(
3 @ToolParam(description = "Customer's email address") String email) {
4 return customers.getOrDefault(email,
5 new Customer(null, "Unknown", email, "Not Found"));
6 }
7 }
```

**Key:** Description tells AI when to call this tool

# Multiple Tools in One Class

```
1 @Component
2 public class CustomerTools {
3 // ... findCustomerByEmail from above ...
4
5 @Tool(description = "List all customers in a membership tier. "
6 + "Valid tiers: Gold, Silver, Bronze")
7 public List<Customer> getCustomersByTier(
8 @ToolParam(description = "Membership tier") String tier) {
9 return customers.values().stream()
10 .filter(c -> c.tier().equalsIgnoreCase(tier))
11 .toList();
12 }
13 }
```

**Register multiple tool classes:** `.defaultTools(weatherTools, customerTools)`

# Tool Design Best Practices

- **Clear descriptions** - Tell AI *when* to use this tool
- **Document parameters** - Use `@ToolParam` with format examples
- **Handle errors gracefully** - Return error info, don't throw
- **Keep tools focused** - One tool, one job

# Good @Tool Description Examples

```
1 @Tool(description = "Get customer by email. Returns name, email, tier if found.")
2
3 @Tool(description = "Calculate order total including tax."
4 + "Use when user asks about pricing or totals.")
5
6 @Tool(description = "Send email notification to a user."
7 + "Returns success/failure status. Use for confirmations.")
```

**Key:** Describe *when* the AI should call the tool

# Security Considerations

- **Authentication** - Verify user context
- **Authorization** - Check permissions
- **Input validation** - Sanitize all inputs
- **Audit logging** - Track tool calls

```
1 @Tool(description = "Get order details by ID")
2 public OrderResponse getOrder(
3 @ToolParam(description = "Order ID") String orderId,
4 ToolContext toolContext) {
5 String userId = (String) toolContext.getContext().get("userId");
6 securityService.checkPermission(userId, "orders:read");
7 return orderService.getOrder(orderId);
8 }
```

# Student Exercise: Custom Tools

Time: 20 minutes

1. **Choose** a domain (e-commerce, HR, etc.)
2. **Create** a Tools class with `@Component`
3. **Add** `@Tool` methods with clear descriptions
4. **Use** `@ToolParam` for parameter documentation
5. **Register** with `.defaultTools()` and test!

# Tool Challenge Ideas

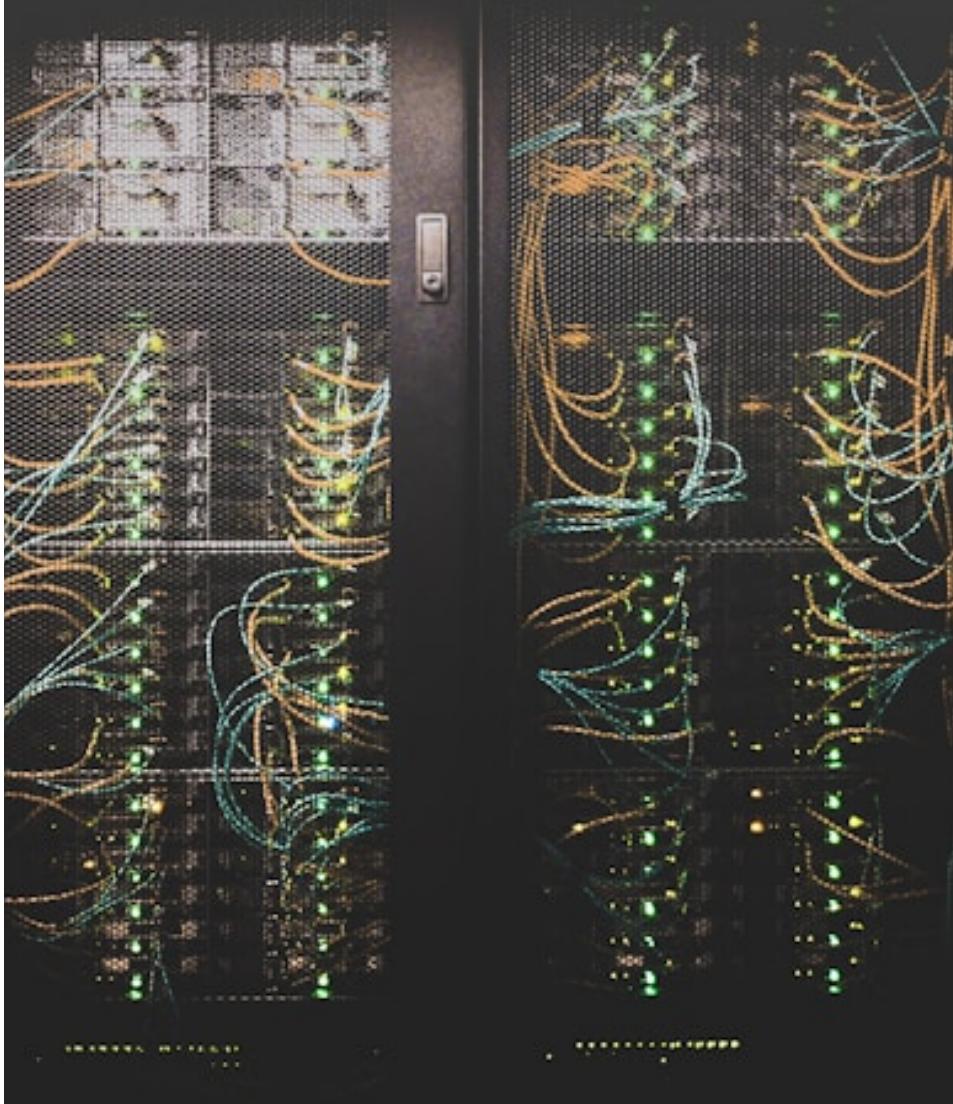
- **Calculator** - Math operations with unit conversion
- **Time tools** - Timezone conversion, date formatting
- **Currency exchange** - Convert between currencies
- **Order management** - createOrder, getStatus, cancelOrder

**Bonus:** Create multiple related `@Tool` methods in one class!

# Part 5: Model Context Protocol (MCP)

## Enhanced Context for Cursor

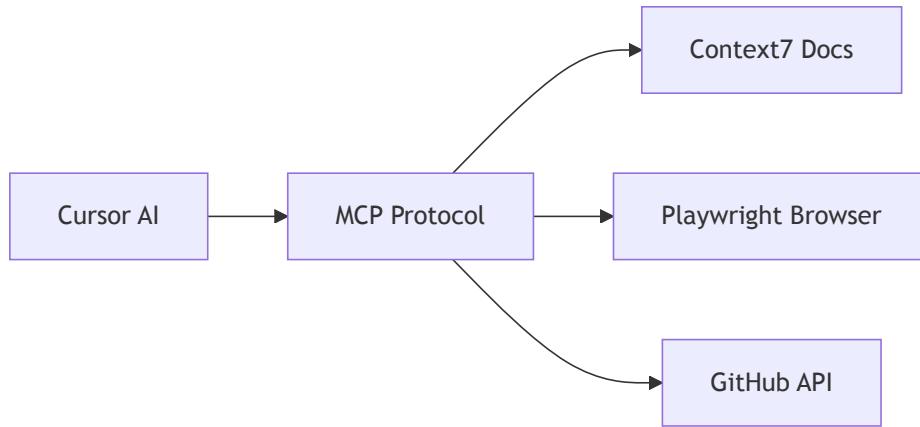
- Protocol for connecting AI to external data
- Real-time database schema awareness
- Tool discovery and dynamic resources



# What is MCP?

- **Protocol** for connecting AI to external data
- **Cursor's MCP support** - Enhanced context in Cursor
- **Standard interface** for tool integration
- **Resources** - Files, databases, APIs

# MCP Architecture



# MCP Benefits

- **Real-time context** - Current database schema
- **Dynamic resources** - Access live data
- **Better suggestions** - More accurate code generation
- **Reduced hallucinations** - Grounded in actual data

# MCP Setup in Cursor

**Step 1:** Cursor → Settings → Features → MCP (or Cmd+Shift+J )

```
1 {
2 "mcpServers": {
3 "context7": {
4 "command": "npx",
5 "args": ["-y", "@upstash/context7-mcp"]
6 }
7 }
8 }
```

**Step 2:** Restart Cursor

# Why Context7?

- **Live documentation** - Always up-to-date library docs
- **No API keys** - Works out of the box
- **Perfect for learning** - Look up Spring AI docs as you code!

**Try:** "Using context7, look up Spring AI ChatClient documentation"

# Available MCP Servers

- `@upstash/context7-mcp` - Live library documentation
- `@anthropic/mcp-server-fetch` - Web page fetching
- `@anthropic/mcp-server-playwright` - Browser automation
- `@anthropic/mcp-server-github` - GitHub API integration
- `@anthropic/mcp-server-filesystem` - Local file access

**Explore more:** [github.com/modelcontextprotocol/servers](https://github.com/modelcontextprotocol/servers)

# MCP + Spring AI Development

Use Context7 while building Spring AI apps:

```
1 "Using context7, show me the latest QuestionAnswerAdvisor API"
2
3 "What's the current syntax for @Tool annotations in Spring AI?"
4
5 "How do I configure SimpleVectorStore in Spring AI 1.1.0?"
```

- Get current API docs, not outdated training data
- Verify your code matches latest patterns
- Learn new features as you build

# MCP Use Cases

- **Documentation lookup** - Current API references while coding
- **Code generation** - Accurate, up-to-date patterns
- **Browser testing** - Interact with running apps (Playwright)
- **Multi-library projects** - Get docs for any dependency

# Student Exercise: MCP Exploration

Time: 10 minutes

1. **Open** Cursor Settings → Features → MCP
2. **Add** Context7 MCP server
3. **Restart** Cursor
4. **Test:** "Using context7, look up Spring AI RAG documentation"

# MCP Exploration Questions

- How do answers differ with vs without Context7?
- Can you get current Spring AI 1.1.0 API details?
- Does Cursor generate more accurate code?
- What other documentation would be useful?

**Try:** "Using context7, what's new in Spring AI 1.1.0?"



# Part 6: Legacy Modernization with AI

## Applying AI Patterns to Legacy Code

- AI-assisted code analysis
- Incremental refactoring strategies
- Adding AI capabilities to existing systems

# Legacy Code Analysis

```
1 // Legacy OrderService (Spring Boot 2.7)
2 @Service
3 public class OrderService {
4 @Autowired private OrderRepository orderRepo;
5 @Autowired private UserRepository userRepo;
6
7 public void processOrder(Long orderId) throws Exception {
8 Order order = orderRepo.findById(orderId).orElseThrow();
9 // Complex business logic, no error handling, hard to test
10 }
11 }
```

# AI Identifies Issues

- Field injection anti-pattern
- Missing validation
- Poor error handling
- No transaction management
- Opportunities for AI enhancement

**Prompt:** "Analyze this legacy OrderService and create a migration plan"

# Step 1: Constructor Injection

```
1 public OrderService(OrderRepository orderRepo, UserRepository userRepo) {
2 this.orderRepo = orderRepo;
3 this.userRepo = userRepo;
4 }
```

## Step 2: Proper Error Handling

```
1 public Order processOrder(Long orderId) {
2 return orderRepo.findById(orderId)
3 .map(this::validateAndProcess)
4 .orElseThrow(() -> new OrderNotFoundException(orderId));
5 }
```

# Step 3: Add AI Capabilities

```
1 public OrderAnalysisReport analyzeOrder(Long orderId) {
2 Order order = getOrder(orderId);
3
4 return chatClient.prompt()
5 .system("Analyze order for risks")
6 .user(toJson(order))
7 .call()
8 .entity(OrderAnalysisReport.class);
9 }
```

# Student Exercise: Legacy Analysis

**Time:** 10 minutes - Use Extended Thinking to explore:

1. "What are the main testing challenges in this legacy codebase?"
2. "How would you prioritize testing improvements?"
3. "What risks should be considered when adding tests?"

# AI-Powered Modernization

- Identify code smells and anti-patterns
- Suggest refactoring strategies
- Generate tests for legacy code
- Add AI capabilities to existing features

**AI suggests:** RAG for policy lookup, function calling for external services

# Wrap-Up & Next Steps

Course Completion

# Spring AI Decision Tree

- **Chat interface?** → Use ChatClient with templates
- **Chat with your data?** → Implement RAG pipeline
- **AI execute code?** → Use function calling
- **Enhanced context?** → Configure MCP

# What We Accomplished

- Spring AI application setup
- ChatClient with prompt templates
- RAG pipeline for document Q&A
- Function calling with Spring AI tools
- MCP exploration for enhanced context

# Core Pattern

Spring AI brings AI capabilities with familiar patterns:

`RestTemplate` → `ChatClient`

`JpaRepository` → `VectorStore`

`@Service` → `@Component` with `@Tool` methods

# The Five-Session Arc

- **Session 1:** Cursor fundamentals - Chat, Agent, Composer
- **Session 2:** Mobile development - Kotlin, Jetpack Compose
- **Session 3:** Agentic coding patterns
- **Session 4:** AI-assisted testing - JUnit 5, Mockito, TestContainers
- **Session 5:** Spring AI - RAG, Function Calling, MCP

# Next Steps

- **Apply Spring AI** to your projects
- **Experiment** with different AI providers
- **Build** custom tools for your domain
- **Stay updated** on Spring AI releases

# Resources

- [Spring AI Docs](#)
- [Spring AI GitHub](#)
- [Spring AI Examples](#)
- [MCP Specification](#)

# Questions & Discussion

Spring AI • RAG • Function Calling • MCP

# Thank You!

## Building AI-Powered Java Apps with Spring AI

**You're now equipped to build intelligent Java applications!**

Ready for the labs? Let's build with Spring AI!