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 Maven Dependencies

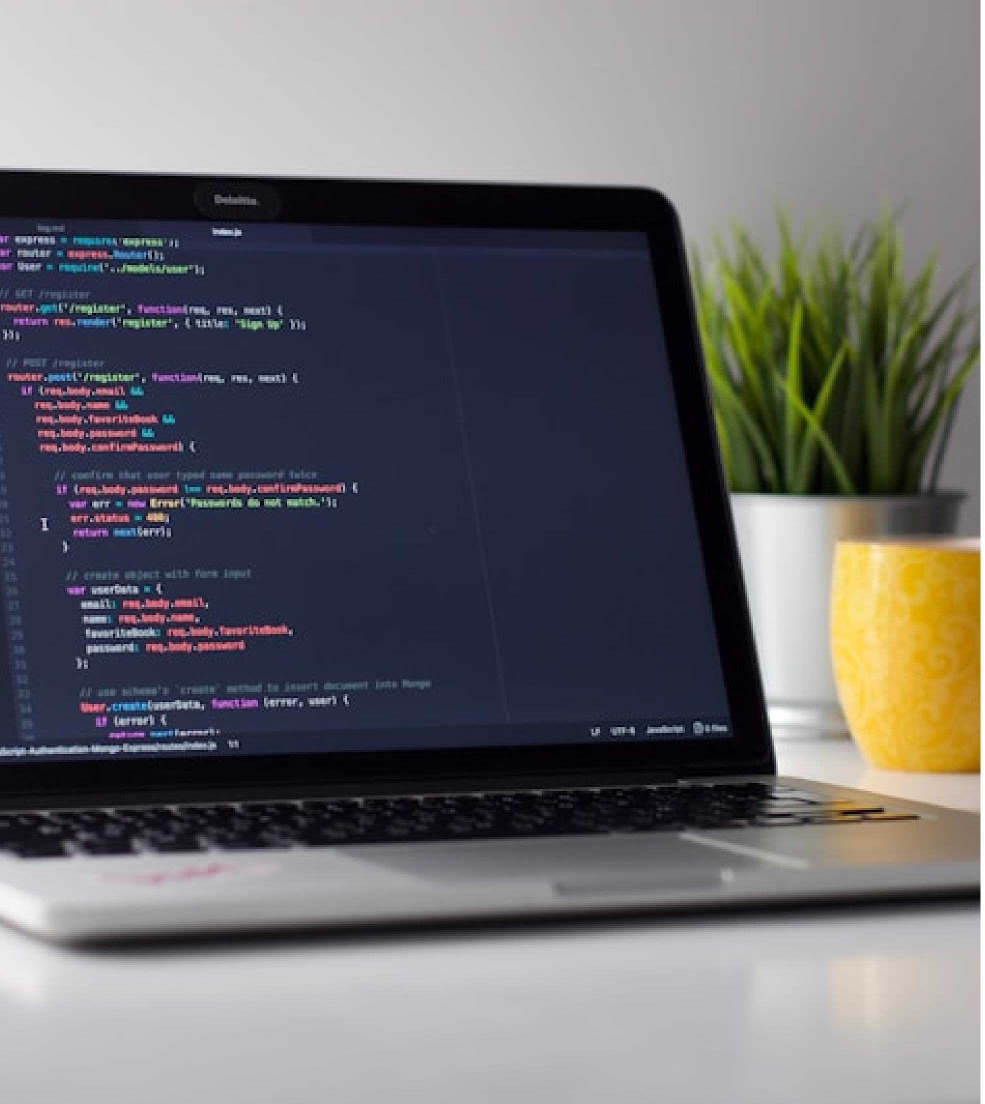
```xml
<parent>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>3.5.7</version>
</parent>

<dependencies>
    <dependency>
        <groupId>org.springframework.ai</groupId>
        <artifactId>spring-ai-starter-model-openai</artifactId>
    </dependency>
</dependencies>
```

Spring AI Configuration

```properties
# application.properties
spring.ai.openai.api-key=${OPENAI_API_KEY}
spring.ai.openai.chat.options.model=gpt-4o
spring.ai.openai.chat.options.temperature=0.7
```

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



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

```java
@RestController
@RequestMapping("/api/chat")
public class ChatController {

    private final ChatClient chatClient;

    public ChatController(ChatClient.Builder builder) {
        this.chatClient = builder.build();
    }

    @GetMapping
    public String chat(@RequestParam String message) {
        return chatClient.prompt()
            .user(message)
            .call()
            .content();
    }
}
```

ChatClient Features

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

Agent Mode Prompt:

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

System Prompts

```java
@GetMapping("/expert")
public String expertChat(@RequestParam String topic) {
    return chatClient.prompt()
        .system("""
            You are an expert software architect
            specializing in Spring Boot applications.
            Provide concise, practical advice.
            """)
        .user("How do I implement " + topic)
        .call()
        .content();
}
```

Structured Responses

```java
record BookReview(String title, int rating, String summary) {}

@GetMapping("/review")
public BookReview getBookReview(@RequestParam String book) {
    return chatClient.prompt()
        .user("Write a review of the book: " + book)
        .call()
        .entity(BookReview.class);
}
```

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

Prompt Templates

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

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

Key Points:

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

Using Prompt Templates

```java
@GetMapping("/joke")
public String tellJoke(
    @RequestParam String topic,
    @RequestParam(defaultValue = "funny") String style) {

    return chatClient.prompt()
        .user(u -> u.text(
            "classpath:/prompts/joke.st",
            Map.of("topic", topic, "style", style)
        ))
        .call()
        .content();
}
```

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

```java
@Component
public class DocumentLoader implements CommandLineRunner {

    private final VectorStore vectorStore;

    public DocumentLoader(VectorStore vectorStore) {
        this.vectorStore = vectorStore;
    }

    @Override
    public void run(String... args) {
        Resource resource = new ClassPathResource("documents/policy.txt");
        TextSplitter splitter = new TokenTextSplitter();
        List<Document> documents = splitter.split(new TextReader(resource).get());
```

Document Ingestion (continued)

```
            // Generate embeddings and store
            vectorStore.add(documents);
            log.info("Loaded {} documents", documents.size());
        }
    }
```

What Happens:

1. Load `policy.txt` from classpath
2. Split into ~500 token chunks
3. Generate embeddings (via OpenAI)
4. Store vectors in SimpleVectorStore

SimpleVectorStore (In-Memory)

```java
@Configuration
public class VectorStoreConfig {

    @Bean
    public VectorStore vectorStore(EmbeddingModel embeddingModel) {
        return new SimpleVectorStore(embeddingModel);
    }
}
```

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

```java
@Bean
public VectorStore vectorStore(JdbcTemplate jdbc, EmbeddingModel model) {
    return new PgVectorStore(jdbc, model);
}
```

RAG Controller Setup

```java
@RestController
@RequestMapping("/api/rag")
public class RagController {

    private final ChatClient chatClient;
    private final VectorStore vectorStore;

    public RagController(ChatClient.Builder builder, VectorStore vectorStore) {
        this.chatClient = builder.build();
        this.vectorStore = vectorStore;
    }
```

RAG Query Implementation

```java
        @GetMapping("/query")
        public String query(@RequestParam String question) {
            // Search for similar documents
            List<Document> similarDocs = vectorStore.similaritySearch(
                SearchRequest.query(question).withTopK(5)
            );

            String context = similarDocs.stream()
                .map(Document::getContent)
                .collect(Collectors.joining("\n\n"));
```

RAG Response Generation

```java
            return chatClient.prompt()
                .system("""
                    Answer the question based ONLY on the provided context.
                    If you cannot answer from the context, say so.
                    Context: {context}
                    """)
                .user(question)
                .call()
                .content();
        }
    }
```

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

```
SearchRequest.query(question)
    .withTopK(5)                   // Return top 5 matches
    .withSimilarityThreshold(0.7)  // Min similarity
    .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

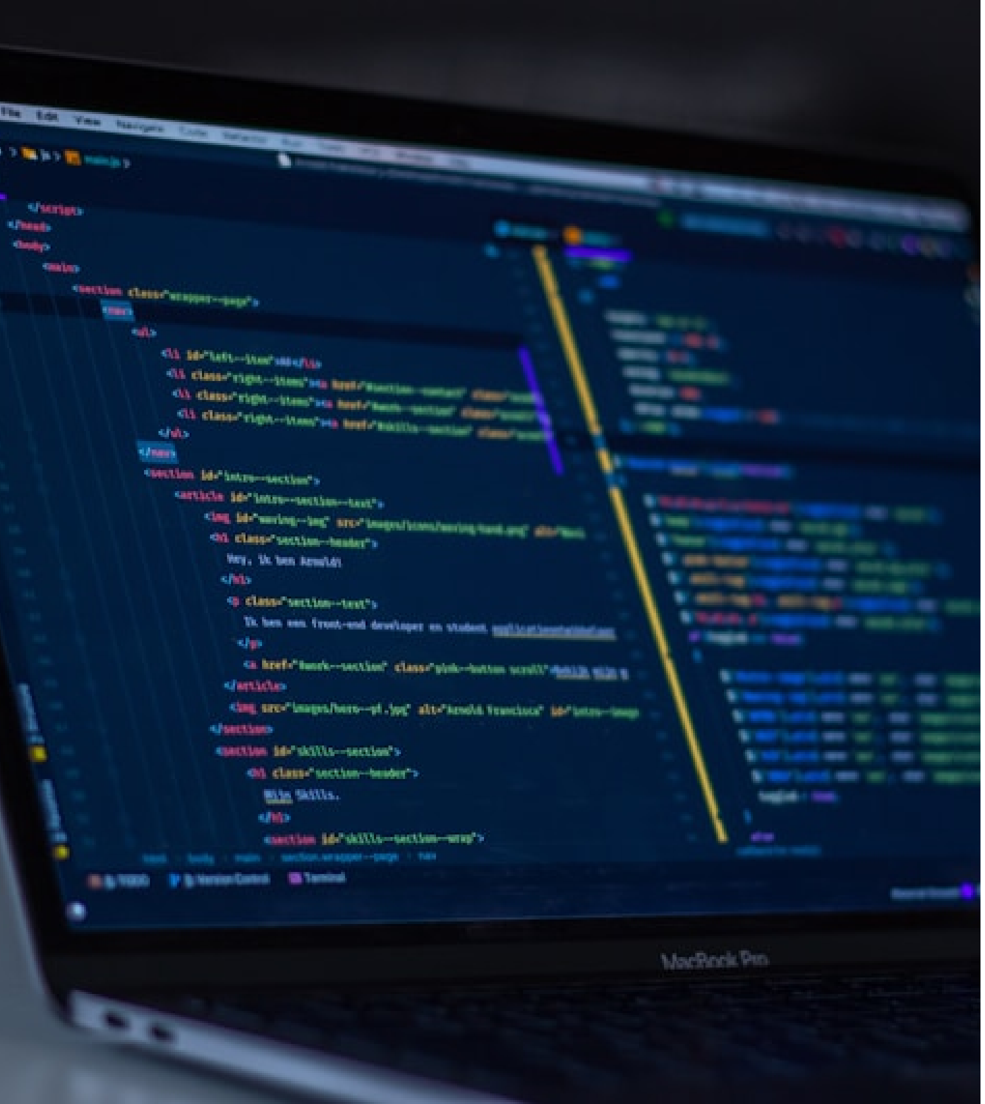
- **Company policies** - HR handbook, procedures
- **Technical docs** - API documentation
- **Knowledge base** - FAQ, troubleshooting

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

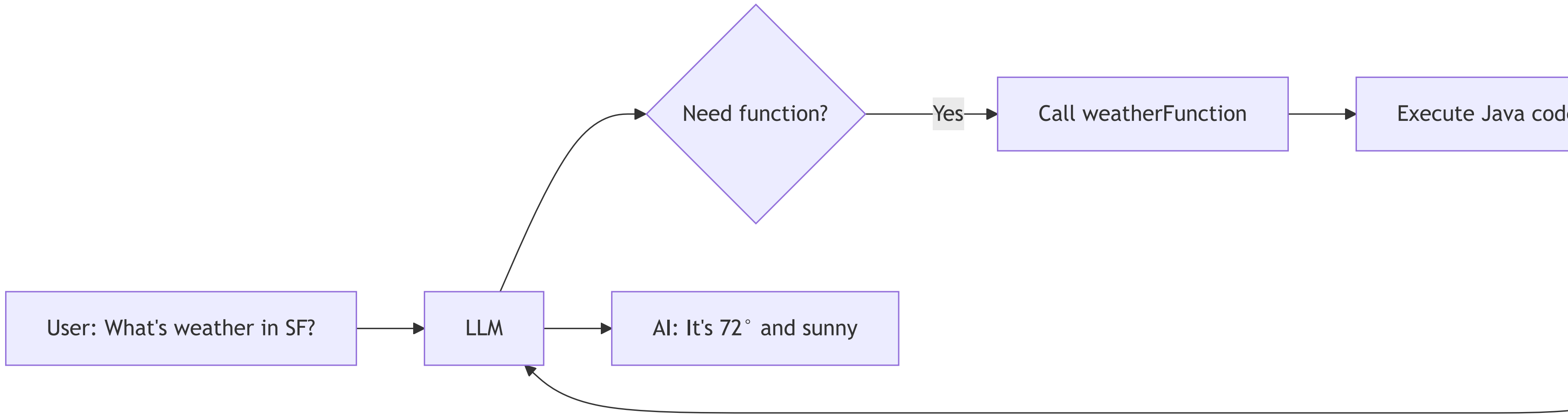
Give AI the Ability to Execute Code

- 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!

Weather Tool: Request/Response

```java
record WeatherRequest(
    @JsonProperty(required = true, value = "location") String location,
    @JsonProperty(required = false, value = "unit") String unit
) {}

record WeatherResponse(
    String location, String temperature, String description
) {}
```

Weather Tool: Function Bean

```java
@Configuration
public class ToolConfig {

    @Bean
    @Description("Get current weather for a location")
    public Function<WeatherRequest, WeatherResponse> weatherFunction() {
        return request -> new WeatherResponse(
            request.location(), "72°F", "Sunny with light clouds"
        );
    }
}
```

Key: `@Description` tells AI when to call this function

Register Functions with ChatClient

```java
@RestController
@RequestMapping("/api/tools")
public class ToolController {

    private final ChatClient chatClient;

    public ToolController(ChatClient.Builder builder,
        Function<WeatherRequest, WeatherResponse> weatherFunction) {
        this.chatClient = builder
            .defaultFunctions(weatherFunction)
            .build();
    }

    @GetMapping("/chat")
    public String chat(@RequestParam String message) {
        return chatClient.prompt().user(message).call().content();
    }
}
```

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 Tool

```java
@Entity
public class User {
    @Id private Long id;
    private String email;
    private String firstName, lastName;
}

public interface UserRepository extends JpaRepository<User, Long> {
    Optional<User> findByEmail(String email);
}
```

User Lookup Function Bean

```java
@Bean
@Description("Find user by email address")
public Function<UserLookupRequest, UserLookupResponse>
    userLookupFunction(UserRepository repo) {

    return request -> repo.findByEmail(request.email())
        .map(u -> new UserLookupResponse(
            u.getFirstName() + " " + u.getLastName(), u.getEmail()
        ))
        .orElse(new UserLookupResponse("Unknown", request.email()));
}
```

Tool Design Best Practices

- **Clear descriptions** - AI needs to understand purpose
- **Strong typing** - Use records for parameters
- **Validation** - Check inputs before executing
- **Error handling** - Return meaningful errors

Good Description Examples

```
@Description("Get user by email. Returns name and email if found.")

@Description("Calculate order total including tax. Returns USD.")

@Description("Send email notification. Returns success status.")
```

Security Considerations

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

```java
@Bean
public Function<OrderLookup, OrderResponse> orderFunction(
    OrderService service, SecurityContext security) {
    return request -> {
        security.checkPermission("orders:read");
        return service.getOrder(request.orderId());
    };
}
```

Student Exercise: Custom Tools

Time: 20 minutes

1. **Choose** a domain (e-commerce, HR, etc.)
2. **Create** request/response records
3. **Implement** function bean with @Description
4. **Register** with ChatClient
5. **Test** with natural language queries

Tool Challenge Ideas

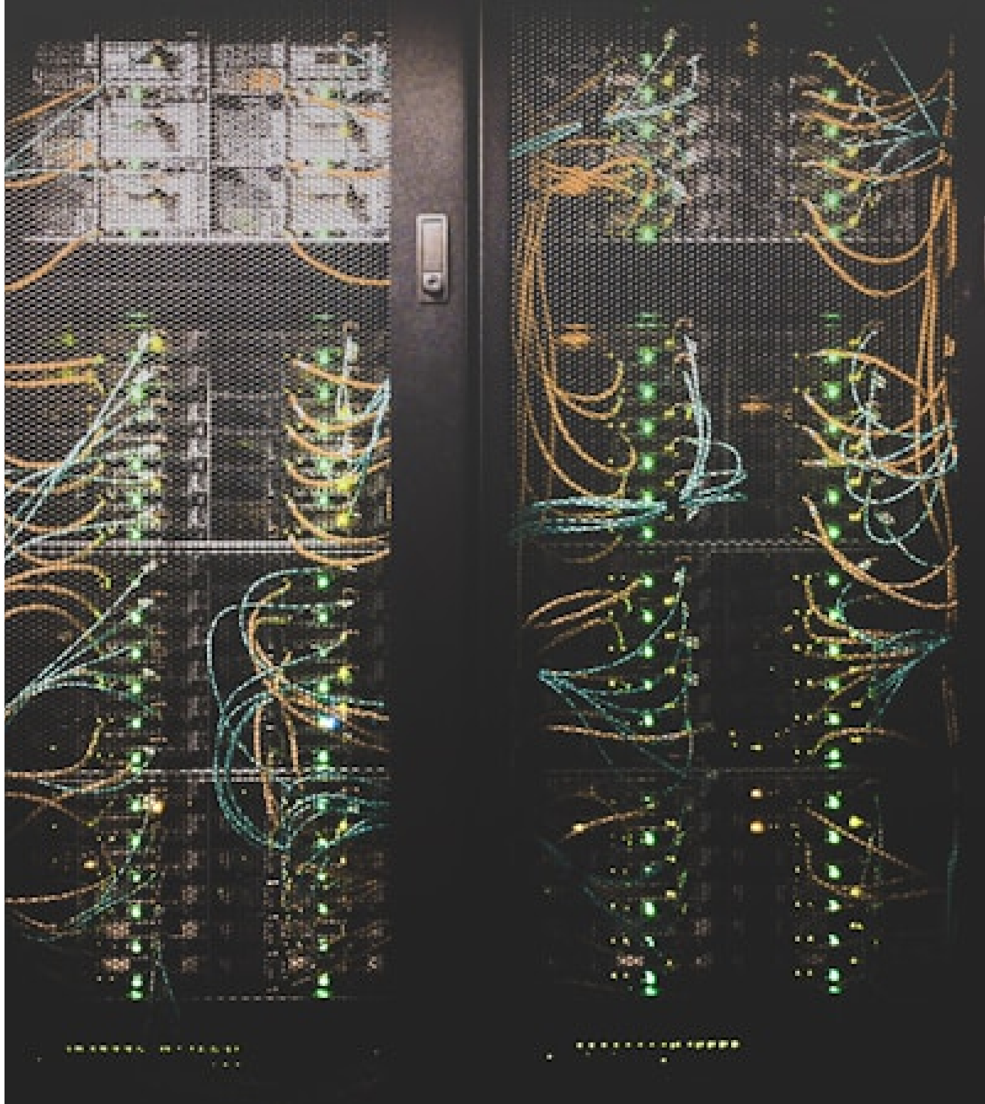
- **Calculator** - Math operations
- **Time converter** - Timezones, formats
- **Currency exchange** - Convert currencies
- **Database query** - Look up records

Bonus: Create related functions (createOrder + getOrderStatus + cancelOrder)

Part 5: Model Context Protocol (MCP)

Enhanced Context for Cursor

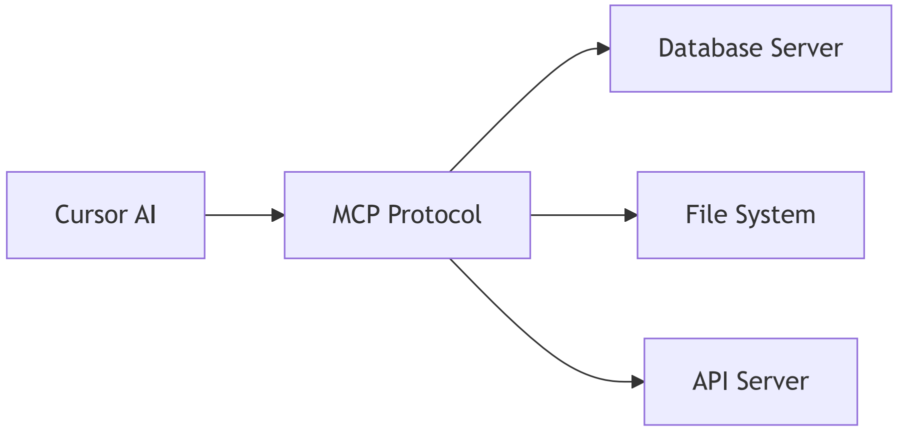
- Protocol for connecting AI to external data
- Real-time database schema awareness
- Tool discovery and dynamic resources



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

```
{
  "mcpServers": {
    "database": {
      "command": "npx",
      "args": ["-y", "@modelcontextprotocol/server-postgres",
               "postgresql://localhost/mydb"]
    }
  }
}
```

Step 2: Restart Cursor

Available MCP Servers

Official:

- `server-postgres` - PostgreSQL
- `server-filesystem` - File access
- `server-github` - GitHub API

Community:

- Jira, Slack, Notion integrations
- Cloud providers (AWS, GCP, Azure)

Spring AI + MCP

Current Approach: Use Spring AI functions as MCP tool building blocks

```java
@Bean
public Function<SchemaRequest, SchemaResponse>
    getDatabaseSchema(DataSource dataSource) {
    return request -> {
        // Query information_schema
        // Return table/column details
    };
}
```

MCP Use Cases

- **Database schema** - Generate accurate SQL, create JPA entities
- **API documentation** - Correct endpoint usage, auth patterns
- **Codebase navigation** - Find related code, understand structure
- **Real-time data** - Status, metrics, analytics

Student Exercise: MCP Exploration

Time: 10 minutes

1. **Open** Cursor Settings → Features → MCP
2. **Add** an MCP server (filesystem or database)
3. **Test** by asking Cursor about your data
4. **Observe** improved context in responses

MCP Exploration Questions

- How does MCP affect Cursor’s suggestions?
- What queries work better with MCP?
- Can you ask about database structure?
- Does Cursor understand your project better?

Try: "What are the main entities in my database?"

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

```java
// Legacy OrderService (Spring Boot 2.7)
@Service
public class OrderService {
    @Autowired private OrderRepository orderRepo;
    @Autowired private UserRepository userRepo;

    public void processOrder(Long orderId) throws Exception {
        Order order = orderRepo.findById(orderId).orElseThrow();
        // Complex business logic, no error handling, hard to test
    }
}
```

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

```java
public OrderService(OrderRepository orderRepo, UserRepository userRepo) {
    this.orderRepo = orderRepo;
    this.userRepo = userRepo;
}
```

Step 2: Proper Error Handling

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

Step 3: Add AI Capabilities

```java
public OrderAnalysisReport analyzeOrder(Long orderId) {
    Order order = getOrder(orderId);

    return chatClient.prompt()
        .system("Analyze order for risks")
        .user(toJson(order))
        .call()
        .entity(OrderAnalysisReport.class);
}
```

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`

`@Bean` → `@Bean Function`

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!