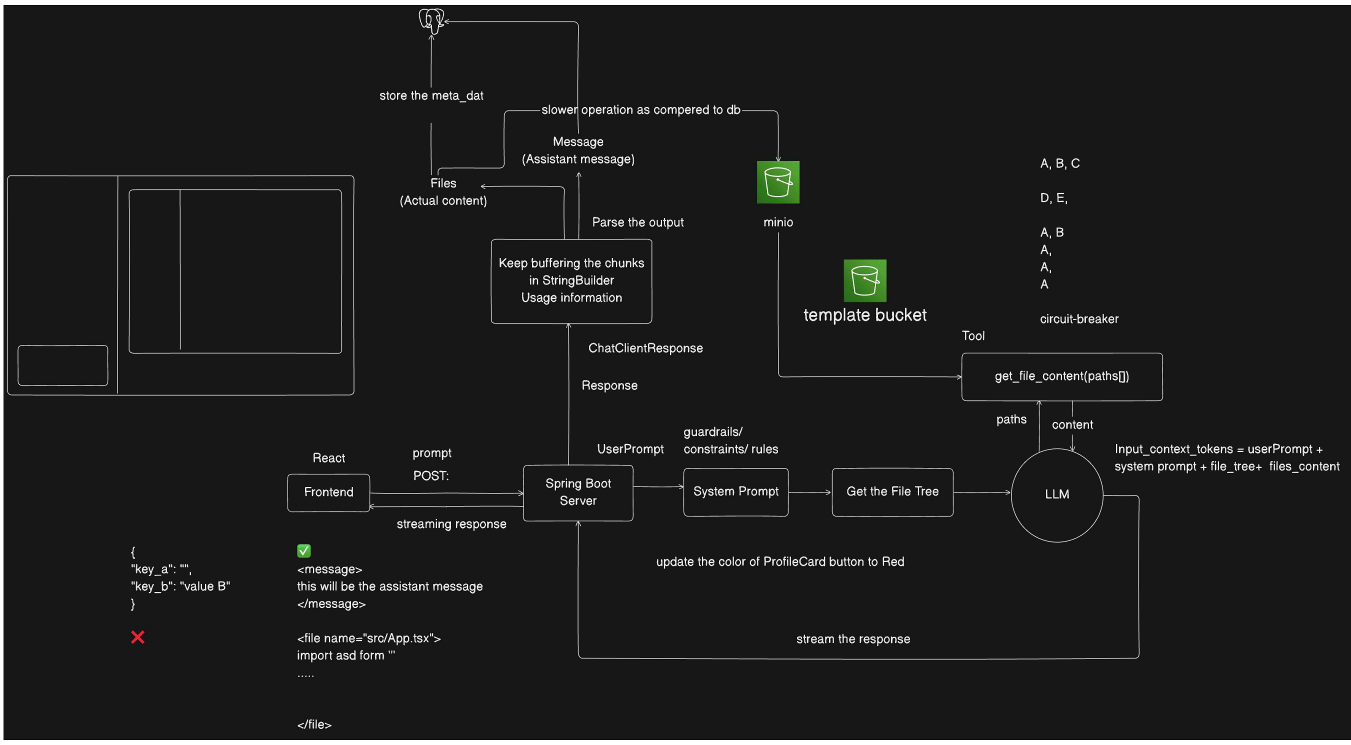
- The below is the Architecture Diagram for code generation with AI.



```
{
"key_a": "",
"key_b": "value B"
}
```

```
<message>
this will be the assistant message
</message>

<file name="src/App.tsx">
import asd form ''
.....


</file>
```

- The flow of the code generation will be as follows:
  - First frontend will send the request to backend & backend will ***stream*** the result, so that frontend will not have to wait till the LLM generates the code.
    Here, frontend will send the **User Prompt** to the user.
  - At the backend, we'll add our own **System Prompt** to make sure the LLM generates the code within a constraint.
    Generally, we'll defind some ***guardrails/constraint/rules*** for the LLM here.
  - Then it'll not be proper to give the full file list to the LLM, it'll cost a lot of tokens and cost will not be optimized in this way.
    So, we'll give the **File Tree** to LLM, it can easily guess the related file/files from that file tree.
    From that File Tree, LLM can generate the path to the file and make a **Tool Call** to get the exact file.
    The tool method will be something like **get_file_content(path[])**, as LLM may require multiple files, so here **path *array*** has been passed.
  - In this case, remembering past message is not required; and many AI code generation platform doesn't do that.
    If it is required, then we can make a **Tool Call** to check the past message; but it will be done if explicitly specified.

- One thing that can break it, recursive file call by the LLM.
  Like first let say LLM made a Tool Call to fetch files A, B. Then it again made a Tool Call to fetch files A, C. sometimes it'll repeatedly fetch file A.
  It is need to be handled, so we need to implement some **Circuit Breaker** here.
- So, the **Input Context** for the LLM is now
  Input Context = User Prompt + System Prompt + File Tree + File Content
- The generated output will now come to our **Server (Backend).**
  It'll be ***streamed***.
- --
- After that, the Backend will keep buffering the ***chunks*** (Document objects with specific max size).
  it'll also be ***streaming***.
  Along with it, the ***token usage*** will also be stored.
- The LLM will give 2 things, **File Content** (code) + **Message** (assistant message).
  If you see in Lovable.dev, on the prompt size assistant messages comes about what it did. And on the right side actual code is generated.
- The **assistant messages** has to be stored so that User can see the messages in the Chat section (Left side panel)
  Also, the **file contents** should be stored so that User can see those (Right side panel).
- You can store the **File Content** inside Vectoe Database, so that LLM can find the related field and update that.

  But it is a **BAD IDEA**, because lets say we give the prompt about doing something, and that content is also having a file name and is also present inside a file as content. In that case it'll not find the actual thing that needs to be changed; even if it finds, it'll change all the related things that we don't want.

  So, instead of storing File Content inside Vector Database, we'll store these to some storage service like **AWS S3** or **MinIO**. (here we'll use MinIO).

  We can put the Data inside the Database, but its not recommended to put a large amount of Data in the Database.

  So, we'll provide the file to LLM depending upon the Path that LLM had requested through Tool Calling and it can then change the exact thing inside that File.

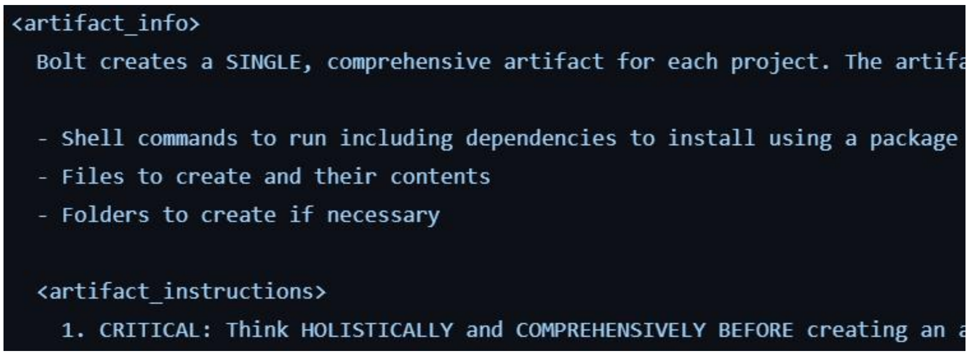
- File's **metadata** will be stored in Postgres database (own database), here **file path, file tree** .. etc will be there.

  But for File Content, it needs to be fetched from minio/S3.
- At the end, the LLM shouldn't create any Project from **SCRATCH**, we'll create one Template which will be edited by LLM.
- --
- Now what will be the streaming format?
  - if the LLM responses with a JSON kind of thing and it is sent back to the frontend via Backend, then frontend will not get to know how to display those JSON contents.
  - So, we'll use **tag format** for the response.

    It'll be having some custom unique tags (that doesn't overlap with HTML tags) and with this our Frontend will get to know about the type of content and load that.

    For example, ***<my_custom_content type="file">***

    and also it'll be having one **closing** tag.

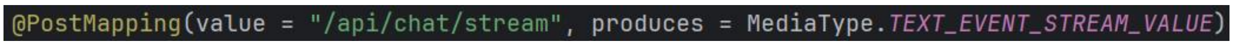
```
<artifact_info>
  Bolt creates a SINGLE, comprehensive artifact for each project. The artifa

   - Shell commands to run including dependencies to install using a package
   - Files to create and their contents
   - Folders to create if necessary

  <artifact_instructions>
    1. CRITICAL: Think HOLISTICALLY and COMPREHENSIVELY BEFORE creating an a
```

    As Spring will not parse these kind of content properly, So we'll make our own **Custom Parser**.

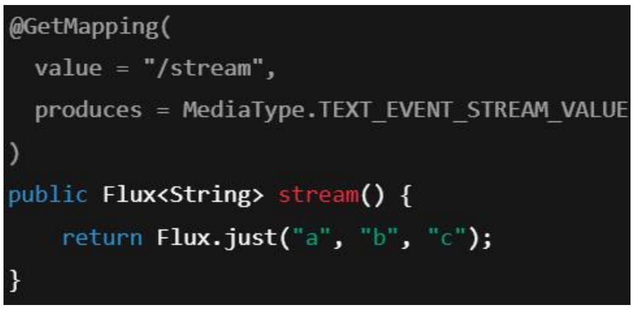
➢ F

Streaming in Spring with Flux

- In the rest mapping like ***@GetMapping***, ***@PostMapping*** ..etc, one value is argument is there which is "**produces**"
  - It is there to select any media type for response.
  - By default its value is *empty array*, which means Spring is free to choose any ***media type***.
  - With **jackson**, the default value of "produces" is **MediaType.APPLICATION_JSON**.
  - We want streaming so we need to add the media type **MediaType.TEXT_EVENT_STREAM_VALUE**



```
@PostMapping(value = "/api/chat/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
```

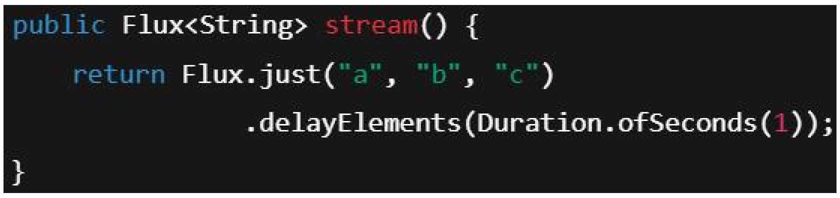
- In case of Spring Reactive, if you write the return type **Flux<String>** then it'll not send the response with *streaming* if you don't mention that **produces** field with TEXT_EVENT_STREAM_VALUE.
  - 

```
@GetMapping(
  value = "/stream",
  produces = MediaType.TEXT_EVENT_STREAM_VALUE
)
public Flux<String> stream() {
    return Flux.just("a", "b", "c");
}
```

you need to write the value of **produces** field and the return type should be **Flux<***some type***>** then only streaming will happen.

But in the above example, it'll not feel like streaming because everything will be fetched very quickly.

To experience streaming you can add some time interval between processing of consecutive elements.



```
public Flux<String> stream() {
    return Flux.just("a", "b", "c")
                .delayElements(Duration.ofSeconds(1));
}
```

  - You might be thinking, we are not calling the **subscribe()** method then how Flux.just("a","b","c") is producing the result.

Spring does this. It calls the ***subscribe()*** method.

In WebFlux, the **framework is the Subscriber**.

```
HTTP request arrives
↓
Spring HandlerAdapter detects return type = Publisher
↓
Spring selects HttpMessageWriter
↓
Spring subscribes to Flux
↓
Flux emits onNext(...)
↓
Writer writes to HTTP response
```

- **ServerSentEvent**
  - What we saw, **Flux<String>** streams only data like “a”, “b”, “c” …etc. But it doesn’t provide some extra information.
  - ***ServerSentEvent*** is just a *wrapper* around those data, and it contains other information (fields) like **id, event, retry, data**

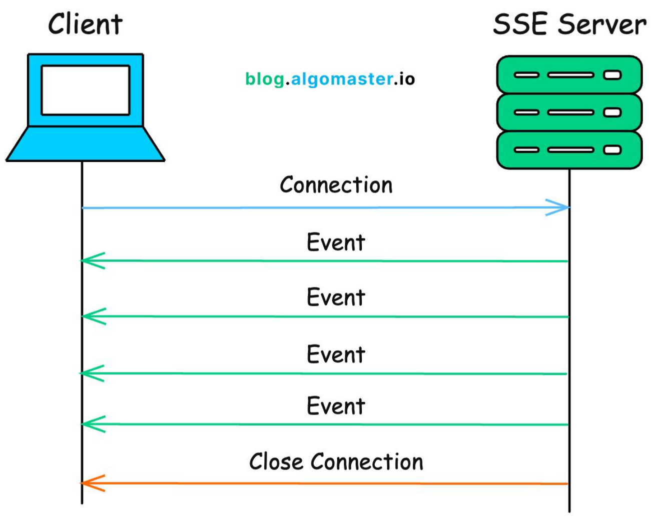
```
id: 42
event: chat
retry: 3000
data: hello
```

  - This is how it is being initialized

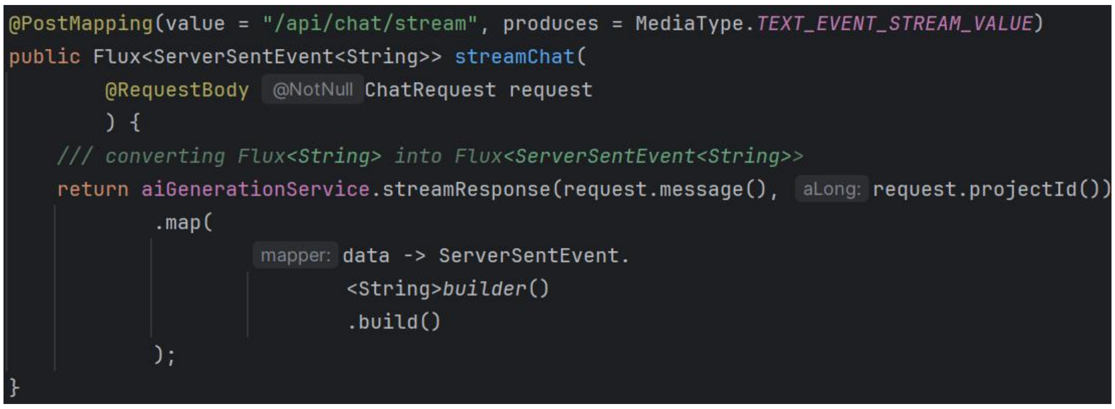
```
ServerSentEvent.builder(data)
    .id("42")
    .event("chat")
    .retry(Duration.ofSeconds(3))
    .comment("typing")
    .build();
```

  - This can be used in the browser side api to listen to the event

```
const es = new EventSource("/stream");

es.addEventListener("chat", e => {
  console.log(e.data);
});
```
(JavaScript)

- **SSE : Server Sent Event**
  - SSE lets the server send a stream of messages to the client, automatically, over HTTP — without the client polling again and again.
  - 

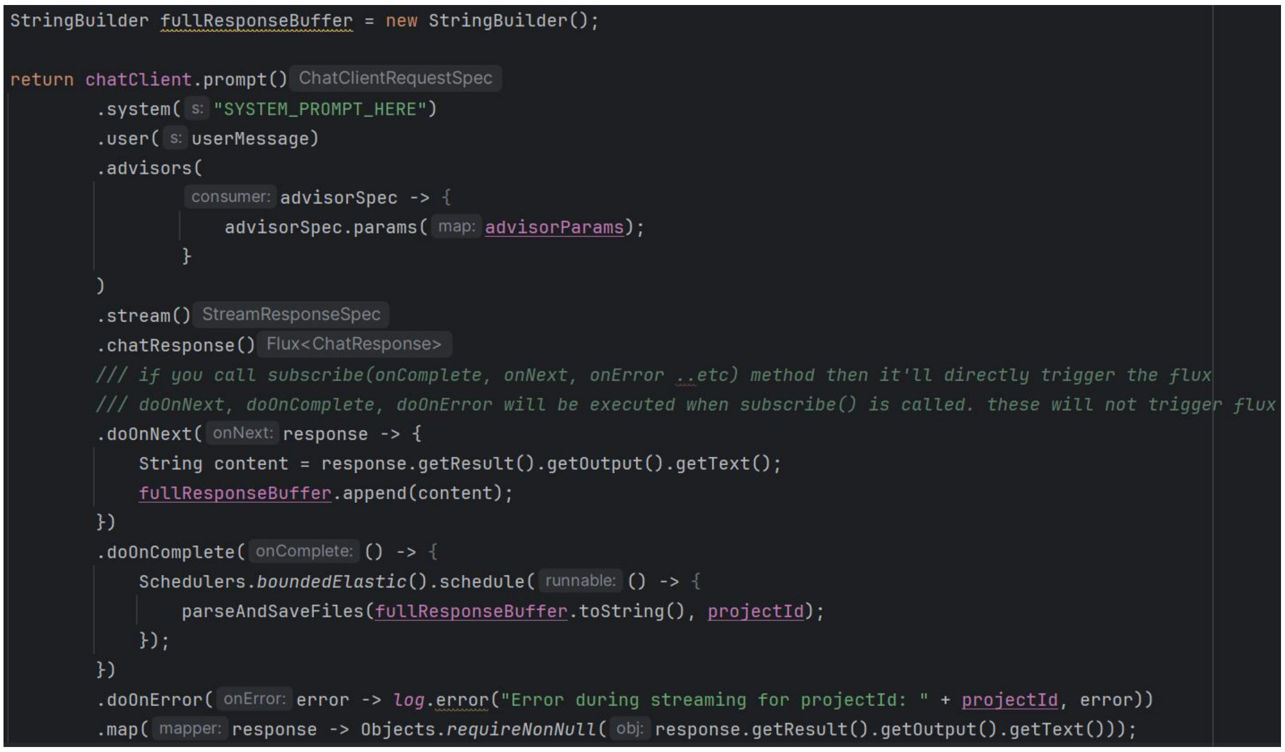
- So, the below is the code to send stream from the controller:



```
@PostMapping(value = "/api/chat/stream", produces = MediaType.TEXT_EVENT_STREAM_VALUE)
public Flux<ServerSentEvent<String>> streamChat(
        @RequestBody  @NotNull ChatRequest request
        ) {
    /// converting Flux<String> into Flux<ServerSentEvent<String>>
    return aiGenerationService.streamResponse(request.message(),  aLong: request.projectId())
            .map(
                    mapper: data -> ServerSentEvent.
                            <String>builder()
                            .build()
            );
}
```

  - Service will return **Flux<String>** from LLM response, and in response we are converting that to **Flux<ServerSentEvent<String>>**

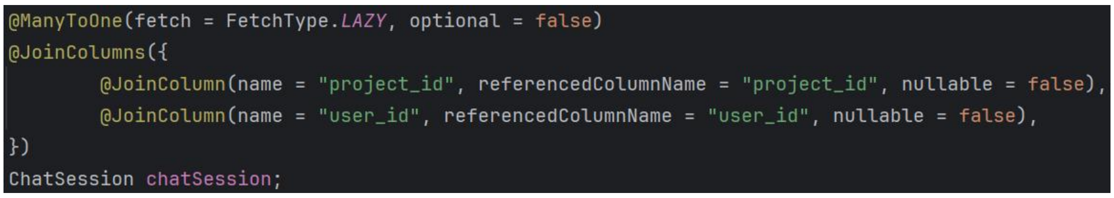
- To stream the response & performing heavy tasks on different thread(Schedulers)



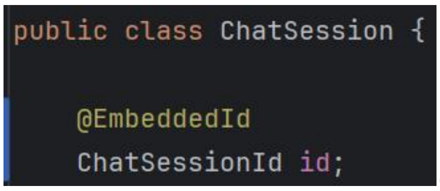
```java
StringBuilder fullResponseBuffer = new StringBuilder();

return chatClient.prompt() ChatClientRequestSpec
        .system( s: "SYSTEM_PROMPT_HERE")
        .user( s: userMessage)
        .advisors(
                consumer: advisorSpec -> {
                    advisorSpec.params( map: advisorParams);
                }
        )
        .stream() StreamResponseSpec
        .chatResponse() Flux<ChatResponse>
        /// if you call subscribe(onComplete, onNext, onError ..etc) method then it'll directly trigger the flux
        /// doOnNext, doOnComplete, doOnError will be executed when subscribe() is called. these will not trigger flux
        .doOnNext( onNext: response -> {
            String content = response.getResult().getOutput().getText();
            fullResponseBuffer.append(content);
        })
        .doOnComplete( onComplete: () -> {
            Schedulers.boundedElastic().schedule( runnable: () -> {
                parseAndSaveFiles(fullResponseBuffer.toString(), projectId);
            });
        })
        .doOnError( onError: error -> log.error("Error during streaming for projectId: " + projectId, error))
        .map( mapper: response -> Objects.requireNonNull( obj: response.getResult().getOutput().getText()));
```

  - **Schedulers.boundedElastic** is used to run that heavy method *parseAndSaveFiles* in a different thread (worker thread).

-
- F
- F
- F
- F

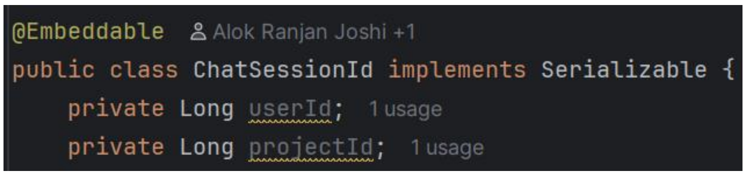
- One **Entity** level NOTE:
  - If one entity contains **Composite Key** (multiple column defines a primary key) and if this entity has ***one-to-many*** relation with another entity |
    this entity: inverse
    another entity: owning
  - In this case, the ***owning*** type will write **@JoinColumns** instead of **@JoinColumn** and add all the columns that combined makes Primary Key (Composite Key).
  - 

```java
@ManyToOne(fetch = FetchType.LAZY, optional = false)
@JoinColumns({
        @JoinColumn(name = "project_id", referencedColumnName = "project_id", nullable = false),
        @JoinColumn(name = "user_id", referencedColumnName = "user_id", nullable = false),
})
ChatSession chatSession;
```

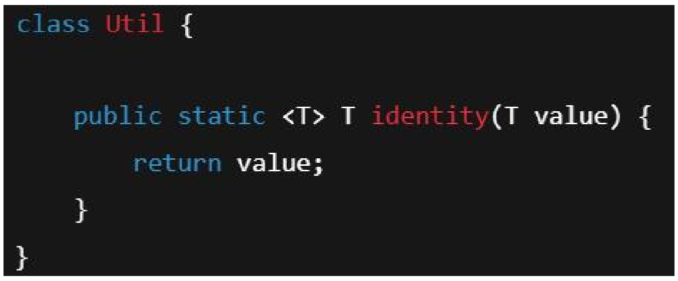


```java
public class ChatSession {

    @EmbeddedId
    ChatSessionId id;
```

(Embedded id i.e. composite key)



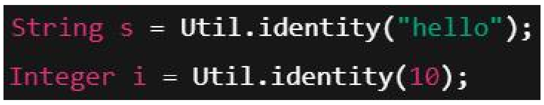
```java
@Embeddable  Alok Ranjan Joshi +1
public class ChatSessionId implements Serializable {
    private Long userId;  1 usage
    private Long projectId;  1 usage
```

- Some basic ***Generics*** rule
  - Consider the below example
    - 

```java
class Util {

    public static <T> T identity(T value) {
        return value;
    }
}
```

Here the *generics* is applied only on the method, not on the class.

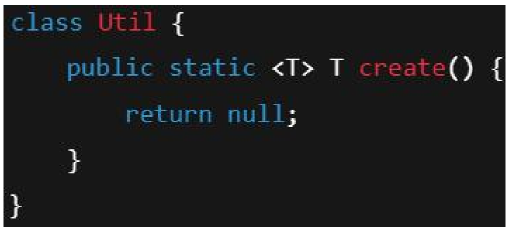
- You cal call it like the below



```java
String s = Util.identity("hello");
Integer i = Util.identity(10);
```

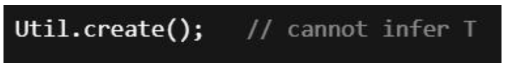
It'll not give any error as it'll take string and integer as the generics by default as we are passing the arguments.

- Now see the below example

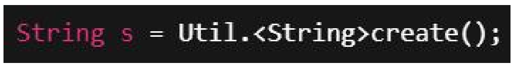


```java
class Util {
    public static <T> T create() {
        return null;
    }
}
```

  - Here no argument is there, so if you call this method directly it’ll give error.



```java
Util.create();   // cannot infer T
```



```java
String s = Util.<String>create();
```

  - It’ll work; it’s the syntax

- F
-