**Data Visualization Backend Description**

1. **Application.properties: -**

It's used to configure various aspects of the application, such as the database connection, logging, and Swagger documentation. Let's go through each property one by one:

1. `server.port=8080`: This property sets the port on which the Spring Boot application will run. In this case, it's configured to run on port 8080.

2. `spring.datasource.driver-class-name=com.mysql.cj.jdbc.Driver`: This property specifies the class name of the JDBC driver used for the database connection. It's set to use the MySQL JDBC driver.

3. `spring.datasource.url=jdbc:mysql://localhost:3306/datavisualization`: This property defines the URL for the MySQL database server. It includes the host (localhost), port (3306), and the name of the database (datavisualization).

4. `spring.datasource.username=root`: This property specifies the username to be used when connecting to the MySQL database. In this case, it's set to "root."

5. `spring.datasource.password=root123`: This property sets the password for the MySQL database connection. It's set to "Nike@9883."

6. `spring.jpa.show-sql=true`: This property enables the display of SQL statements generated by the Spring Data JPA framework in the application logs. When set to "true," it will show SQL statements in the log.

7. `spring.jpa.hibernate.ddl-auto=update`: This property controls how Hibernate, the JPA implementation, manages database schema changes. "update" means that Hibernate will automatically update the database schema based on the entity classes.

8. `spring.jpa.properties.hibernate.dialect=org.hibernate.dialect.MySQL8Dialect`: This property sets the Hibernate dialect for MySQL 8. It tells Hibernate how to generate SQL statements specific to MySQL 8.

9. `logging.level.com.maveric.datavisualization=DEBUG`: This property configures the logging level for a specific package ("com.maveric.datavisualization") to DEBUG. It means that log messages for this package will be displayed at the DEBUG level.

Or

In simple language, the line `logging.level.com.maveric.datavisualization=DEBUG` in configuration file is telling your application how detailed it should be when logging information related to a specific part of your code.

Here's a breakdown:

- `logging.level.com.maveric.datavisualization`: This part specifies which part of your application's codebase should be affected by this configuration. In this case, it's a package or module called "com.maveric.datavisualization."

- `DEBUG`: This part specifies the level of detail for logging. "DEBUG" is one of the logging levels, and it's quite detailed. It means that the application will log a lot of information for the "com.maveric.datavisualization" package, helping developers troubleshoot and understand what's happening in that specific part of the code.

So, in simple terms, this configuration is telling your application to generate detailed logs (at the DEBUG level) for the "com.maveric.datavisualization" part of your code. This can be useful for debugging and monitoring that specific part of your application when it's running.

10. `logging.file.name=/home/thinknxt/logs/dev/datavisualization.logs`: This property defines the file path and name for the log file where application logs will be written.

11. `springdoc.swagger-ui.path=/swagger-ui.html`: This property sets the path at which the Swagger UI documentation for the API will be available. In this case, it will be available at "/swagger-ui.html."

[

Swagger is an open-source framework for designing, building, documenting, and testing RESTful APIs (Application Programming Interfaces). It provides a set of tools and specifications that enable developers to create APIs in a standardized and consistent manner. Here are some key aspects of Swagger:

1. \*\*API Documentation:\*\* Swagger allows developers to create interactive and user-friendly documentation for their APIs. This documentation includes details about the available endpoints, request and response parameters, data types, and example usage. This documentation is often presented in a web-based interface, making it easy for both developers and non-developers to understand and use the API.

2. \*\*Code Generation:\*\* Swagger can generate server and client code in various programming languages based on the API specifications. This helps developers save time and ensures that the code is consistent with the API documentation.

3. \*\*Testing:\*\* Swagger provides tools for testing APIs directly from the documentation. Users can make requests to API endpoints and see the responses without needing to write custom code for testing.

4. \*\*Standardization:\*\* Swagger uses the OpenAPI Specification (formerly known as the Swagger Specification) to define API contracts. This specification is machine-readable and language-agnostic, allowing developers to describe APIs in a way that can be understood by various programming languages and tools.

5. \*\*Integration:\*\* Many development tools and frameworks support Swagger, making it easy to integrate with existing development workflows. This includes support for frameworks like Spring Boot, Express.js, Django, and more.

6. \*\*Versioning:\*\* Swagger supports versioning of APIs. Developers can maintain multiple versions of their APIs, and Swagger documentation can clearly specify which version of the API is being documented.

Overall, Swagger simplifies the process of designing, documenting, and testing APIs, making it an essential tool for developers and organizations building and maintaining web services and RESTful APIs. It helps ensure that APIs are well-documented, consistent, and easy to understand, which is crucial for successful API adoption and usage.]

1. **CrossOrigin: -**

The `@CrossOrigin(origins = "\*")` annotation in Spring Boot is used to enable Cross-Origin Resource Sharing (CORS) support for a specific controller method or an entire controller class. CORS is a security feature implemented by web browsers that restricts web pages from making requests to a different domain than the one that served the web page. This restriction is in place to prevent potential security vulnerabilities, such as Cross-Site Request Forgery (CSRF) and Cross-Site Scripting (XSS) attacks.

When you apply `@CrossOrigin(origins = "\*")` to a controller method or class, it essentially tells the browser that it's acceptable for web pages from any origin (indicated by the `\*`) to make requests to this particular endpoint. This is useful when you want to allow cross-origin requests to access your API.

Here's a breakdown of what the annotation does:

- `@CrossOrigin`: This is an annotation provided by Spring Framework to control CORS behavior.

- `origins = "\*"`: This part of the annotation specifies the allowed origins. In this case, the `\*` wildcard means that requests from any origin are allowed. You can replace `\*` with specific origins (e.g., `https://example.com`) to allow requests only from specific domains.

When you apply `@CrossOrigin(origins = "\*")` to a controller method or class, it overrides the default CORS configuration and allows cross-origin requests to access that particular endpoint. This can be useful when you're building a RESTful API and need to make it accessible to web applications hosted on different domains.

1. **RequestMapping: -**

The `@RequestMapping("/user")` annotation in a Spring Boot controller is used to define a mapping between an HTTP request and a specific controller method. In this case, it specifies that the annotated controller class or method will handle HTTP requests that have a URL path starting with "/user."

Here's how it works:

- `@RequestMapping("/user")`: This annotation is applied to a class or method within a Spring Boot controller. It indicates that the annotated class or method can handle HTTP requests that match the URL path "/user" or paths that start with "/user." For example, it will handle requests like "/user/profile" or "/user/create."

You can use this annotation at the class level to define a base URL for all methods within the controller, or you can use it at the method level to specify a more specific URL path for that particular method.

1. private static final Logger logger = LoggerFactory.getLogger(UserController.class);

The line `private static final Logger logger = LoggerFactory.getLogger(UserController.class);` is used to create a logger instance in a Java class, specifically in the `UserController` class in your example. This is a common practice in Java and is used for logging messages and information about the application's behavior.

Here's what this line does:

`private`: This keyword specifies that the logger is a private field, meaning it can only be accessed within the `UserController` class.

`static`: This keyword indicates that the logger is a static field. Static means that it belongs to the class itself, rather than an instance of the class. This allows you to use the logger without creating an instance of the class.

`final`: This keyword means that the logger cannot be reassigned to another logger instance once it is initialized.

`Logger logger`: This declares a variable named `logger` of type `Logger`. The `Logger` class is typically provided by a logging framework like SLF4J (Simple Logging Facade for Java).

`LoggerFactory.getLogger(UserController.class)`: This is the part of the code that initializes the logger. It calls a method from the `LoggerFactory` class to create a logger instance associated with the `UserController` class. This logger will be used to log messages and events related to the `UserController` class.

With this logger in place, you can use it to log various messages and information in your `UserController` class, which can be helpful for debugging, monitoring, and understanding how your application behaves. For example, you can use methods like `logger.info()`, `logger.debug()`, `logger.error()`, etc., to log different levels of messages.

By logging messages at different levels, you can track the flow of your application and identify issues when they occur. The actual behavior of logging (e.g., where log messages are written and their format) is typically configured through a logging framework like Logback or Log4j, which is integrated with SLF4J.