Here's a comparison of JDK 8, JDK 17, and JDK 21:

**JDK 8**

* **Release Date**: March 2014
* **LTS (Long-Term Support)**: Yes
* **Key Features**:
  + **Lambda Expressions**: Introduced functional programming to Java, allowing for more concise and expressive code.
  + **Streams API**: Enabled sophisticated data manipulation and processing.
  + **Optional Class**: Helped avoid NullPointerException by providing a container that may or may not contain a non-null value.
  + **New Date and Time API**: Replaced the old java.util.Date with a more consistent and flexible API in java.time [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).
* **Pros**:
  + High stability and extensive community support.
  + Large ecosystem of libraries built with Java 8 compatibility.
* **Cons**:
  + Lacks modern enhancements, security patches, and optimizations found in newer versions.
  + No modularity, leading to a larger runtime and application footprint [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).

**JDK 17**

* **Release Date**: September 2021
* **LTS**: Yes
* **Key Features**:
  + **Sealed Classes**: Restricted which other classes or interfaces may extend or implement them.
  + **Pattern Matching for instanceof**: Simplified common coding patterns.
  + **New macOS Rendering Pipeline**: Improved performance on macOS.
  + **Enhanced Pseudo-Random Number Generators**: Provided new interfaces and implementations for random number generation [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).
* **Pros**:
  + Modern enhancements and optimizations.
  + Improved performance and security features.
* **Cons**:
  + Requires migration effort from older versions due to significant changes and deprecations [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).

**JDK 21**

* **Release Date**: September 2023
* **LTS**: Yes
* **Key Features**:
  + **Virtual Threads**: Simplified writing, maintaining, and observing high-throughput concurrent applications.
  + **Record Patterns**: Enhanced pattern matching capabilities.
  + **Foreign Function & Memory API**: Provided a safe and efficient way to access foreign memory and call foreign functions.
  + **Vector API**: Enabled the writing of complex vector computations that compile to optimal vector hardware instructions [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).
* **Pros**:
  + Latest features and improvements.
  + Enhanced performance and modern APIs.
* **Cons**:
  + As the newest LTS version, it may have less community support compared to older LTS versions [[1]](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/).

Choosing the right JDK version depends on your project's specific needs, such as stability, performance, and the necessity for modern features. If you need any more details or have specific questions, feel free to ask!

**References**

[1] [Java 8 vs Java 11 vs Java 17 vs Java 21: A Comprehensive Comparison](https://www.guvi.in/blog/java-8-vs-java-11-vs-java-17-vs-java-21-comparison/)

**Java Version Comparison Table**

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Feature** | **Java 8** | **Java 11** | **Java 17** | **Java 21** |
| **LTS** | Yes | Yes | Yes | Yes |
| **Lambda Expressions** | Yes | Yes | Yes | Yes |
| **Modularity (Jigsaw)** | No | Yes | Yes | Yes |
| **HTTP Client API** | No | Yes | Yes | Yes |
| **Switch Expressions** | No | No | Yes | Yes |
| **Pattern Matching** | No | No | Yes | Yes |
| **Virtual Threads** | No | No | No | Yes (Preview) |
| **String Templates** | No | No | No | Yes (Preview) |
| **Sealed Classes** | No | No | Yes | Yes |
| **Performance Enhancements** | Basic | Improved | Improved | Advanced |

Migrating a Spring Boot 3.0 application from JDK 8 to JDK 21 involves several steps and considerations to ensure compatibility and leverage new features. Here are some key changes and steps you might need to follow:

**1. Update Dependencies**

* **Spring Boot Version**: Ensure your Spring Boot version is updated to 3.0 or later. Spring Boot 3.0 is compatible with JDK 17 and above.
* **Library Updates**: Update all dependencies to versions compatible with JDK 21. Check for any deprecated libraries and replace them with supported alternatives.

**2. Code Changes**

* **Lambda Expressions and Streams**: Review and refactor code to use lambda expressions and streams more effectively, as these features were introduced in JDK 8 but can be optimized further in newer versions.
* **Optional Class**: Ensure proper usage of the Optional class to avoid NullPointerException.
* **Pattern Matching**: Utilize pattern matching for instanceof introduced in JDK 17 to simplify type checks.
* **Records and Sealed Classes**: Consider using records for immutable data structures and sealed classes for restricted class hierarchies.

**3. Concurrency Enhancements**

* **Virtual Threads**: Leverage virtual threads introduced in JDK 21 for better concurrency management. This can simplify writing high-throughput concurrent applications.
* **Structured Concurrency**: Use the structured concurrency API to manage and observe virtual threads more effectively.

**4. Foreign Function & Memory API**

* **Interoperability**: Utilize the Foreign Function & Memory API for safe and efficient access to foreign memory and calling foreign functions, which can be useful for integrating with native libraries.

**5. Vector API**

* **Performance Optimization**: Use the Vector API for complex vector computations that compile to optimal vector hardware instructions, enhancing performance.

**6. Testing and Validation**

* **Compatibility Testing**: Conduct thorough testing to ensure all parts of your application work correctly with JDK 21. Pay special attention to any deprecated or removed features.
* **Performance Testing**: Benchmark your application to identify any performance improvements or regressions.

**7. Build and Deployment**

* **Build Tools**: Update your build tools (e.g., Maven or Gradle) to versions that support JDK 21.
* **Containerization**: If using Docker, update your Dockerfile to use a base image compatible with JDK 21.

**8. Documentation and Training**

* **Documentation**: Update your project documentation to reflect changes made during the migration.
* **Training**: Ensure your development team is familiar with new features and best practices introduced in JDK 21.

Migrating to a newer JDK version can bring significant performance and security improvements, but it requires careful planning and execution. If you need more detailed guidance or run into specific issues, feel free to ask!