Red Hat Data Grid 8.2

Cache Encoding and Marshalling

Encoding Data Grid caches and marshalling Java objects
Encoding Data Grid caches and marshalling Java objects
Abstract

Data Grid caches can store keys and values in different encodings. This document describes how Data Grid encodes data for remote and embedded caches and explains how to use various media types with your applications. Additionally this guide describes how to use the Data Grid implementation of the ProtoStream API to marshall Java objects as Protocol Buffers (Protobuf). You can also find information about adding ProtoStream annotations to your Java classes and then generating Protobuf schema for marshalling capabilities as well as indexed queries.
Table of Contents

RED HAT DATA GRID ............................................................................................................. 4
DATA GRID DOCUMENTATION ............................................................................................... 5
DATA GRID DOWNLOADS ......................................................................................................... 6
MAKING OPEN SOURCE MORE INCLUSIVE ................................................................................ 7

CHAPTER 1. CONFIGURING CACHE ENCODING ....................................................................... 8
  1.1. CACHE ENCODING ........................................................................................................... 8
      Remote caches .................................................................................................................. 8
      Embedded caches .......................................................................................................... 8
  1.2. PROTOBUF CACHE ENCODING ...................................................................................... 8
      Interoperability ............................................................................................................... 9
      Queries ........................................................................................................................... 9
      Custom types ................................................................................................................ 9
      1.2.1. Encoding caches as ProtoStream ........................................................................... 10
  1.3. TEXT-BASED CACHE ENCODING .................................................................................... 10
      1.3.1. Clients and text-based encoding .......................................................................... 11
      Hot Rod clients .............................................................................................................. 11
      REST clients ................................................................................................................ 11
  1.4. MARSHALLED JAVA OBJECTS ....................................................................................... 11
      1.4.1. Clients and marshalled objects ............................................................................. 12
  1.5. PLAIN OLD JAVA OBJECTS (POJO) ................................................................................ 12
      1.5.1. Clients and POJOs ............................................................................................... 13
      Hot Rod clients .............................................................................................................. 13
      REST clients ................................................................................................................ 13
  1.6. ADDING JARS TO DATA GRID SERVER INSTALLATIONS .............................................. 13

CHAPTER 2. MARSHALLING CUSTOM OBJECTS WITH PROTOSTREAM ................................. 14
  2.1. PROTOSTREAM MARSHALLING ..................................................................................... 14
      2.1.1. ProtoStream types ................................................................................................. 14
      Additional type collections ............................................................................................ 15
      2.1.2. ProtoStream annotations ....................................................................................... 16
      ProtoField ...................................................................................................................... 16
      ProtoFactory ................................................................................................................. 17
      AutoProtoSchemaBuilder ............................................................................................. 17
      ProtoAdapter ................................................................................................................ 18
      ProtoDoc and ProtoDocs ............................................................................................... 18
      ProtoName .................................................................................................................... 18
      ProtoEnumValue ............................................................................................................ 18
      ProtoReserved and ProtoReservedStatements ................................................................ 18
      ProtoTypeld .................................................................................................................. 18
      ProtoUnknownFieldSet ............................................................................................... 19
  2.2. CREATING SERIALIZATION CONTEXT INITIALIZERS .................................................. 19
      2.2.1. Adding the ProtoStream processor ....................................................................... 19
      2.2.2. Adding ProtoStream annotations to Java classes ................................................ 20
      2.2.3. Creating ProtoStream adapter classes .................................................................. 22
      2.2.4. Generating serialization context initializers ........................................................ 23
      2.2.5. Registering serialization context initializers ......................................................... 24
      2.2.6. Registering Protobuf schemas with Data Grid Server ......................................... 25
      2.2.7. Manual serialization context initializer implementations ..................................... 26
CHAPTER 3. USING ALTERNATIVE AND CUSTOM MARSHALLER IMPLEMENTATIONS .......................... 28
  3.1. ALLOWING DESERIALIZATION OF JAVA CLASSES .................................................. 28
  3.2. USING JBOSS MARSHALLING ................................................................................. 28
  3.3. USING JAVA SERIALIZATION ............................................................................. 29
  3.4. USING CUSTOM MARSHALLERS ............................................................................ 30

CHAPTER 4. DATA CONVERSION ......................................................................................... 32
  4.1. HOT ROD DATAFORMAT API ................................................................................. 32
  4.2. CONVERTING DATA ON DEMAND WITH EMBEDDED CACHES .............................. 33
RED HAT DATA GRID

Data Grid is a high-performance, distributed in-memory data store.

**Schemaless data structure**
Flexibility to store different objects as key-value pairs.

**Grid-based data storage**
Designed to distribute and replicate data across clusters.

**Elastic scaling**
Dynamically adjust the number of nodes to meet demand without service disruption.

**Data interoperability**
Store, retrieve, and query data in the grid from different endpoints.
DATA GRID DOCUMENTATION

Documentation for Data Grid is available on the Red Hat customer portal.

- Data Grid 8.2 Documentation
- Data Grid 8.2 Component Details
- Supported Configurations for Data Grid 8.2
- Data Grid 8 Feature Support
- Data Grid Deprecated Features and Functionality
DATA GRID DOWNLOADS

Access the Data Grid Software Downloads on the Red Hat customer portal.

NOTE

You must have a Red Hat account to access and download Data Grid software.
MAKING OPEN SOURCE MORE INCLUSIVE

Red Hat is committed to replacing problematic language in our code, documentation, and web properties. We are beginning with these four terms: master, slave, blacklist, and whitelist. Because of the enormity of this endeavor, these changes will be implemented gradually over several upcoming releases. For more details, see our CTO Chris Wright’s message.
CHAPTER 1. CONFIGURING CACHE ENCODING

Find out how to configure Data Grid caches with different media types and how encoding affects the ways you can use Data Grid.

1.1. CACHE ENCODING

Encoding is the format, identified by a media type, that Data Grid uses to store entries (key/value pairs) in caches.

Remote caches
Data Grid Server stores entries in remote caches with the encoding that is set in the cache configuration.

Hot Rod and REST clients include a media type with each request they make to Data Grid Server. To handle multiple clients making read and write requests with different media types, Data Grid Server converts data on-demand to and from the media type that is set in the cache configuration.

If the remote cache does not have any encoding configuration, Data Grid Server stores keys and values as generic `byte[]` without any media type information, which can lead to unexpected results when converting data for clients request different formats.

Use ProtoStream encoding
Data Grid Server returns an error when client requests include a media type that it cannot convert to or from the media type that is set in the cache configuration.

Data Grid recommends always configuring cache encoding with the `application/x-protostream` media type if you want to use multiple clients, such as Data Grid Console or CLI, Hot Rod, or REST. ProtoStream encoding also lets you use server-side tasks and perform indexed queries on remote caches.

Embedded caches
Data Grid stores entries in embedded caches as Plain Old Java Objects (POJOs) by default.

For clustered embedded caches, Data Grid needs to marshal any POJOs to a byte array that can be replicated between nodes and then unmarshalled back into POJOs. This means you must ensure that Data Grid can serialize your POJOs with the ProtoStream marshaller if you do not configure another marshaller.

NOTE

If you store mutable POJOs in embedded caches, you should always update values using new POJO instances. For example, if you store a `HashMap` as a key/value pair, the other members of the Data Grid cluster do not see any local modifications to the Map. Additionally, it is possible that a `ConcurrentModificationException` could occur if the Map instance is updated at the same time that Data Grid is marshalling the object.

Additional resources

- Data Grid ProtoStream API

1.2. PROTOBUF CACHE ENCODING
Protocol Buffers (Protobuf) is a lightweight binary media type for structured data. As a cache encoding, Protobuf gives you excellent performance as well as interoperability between client applications in different programming languages for both Hot Rod and REST endpoints.

Data Grid uses a ProtoStream library to encode caches as Protobuf with the \texttt{application/x-protostream} media type.

The following example shows a Protobuf message that describes a \texttt{Person} object:

```protobuf
message Person {
  optional int32 id = 1;
  optional string name = 2;
  optional string surname = 3;
  optional Address address = 4;
  repeated PhoneNumber phoneNumbers = 5;
  optional uint32 age = 6;
  enum Gender {
    MALE = 0;
    FEMALE = 1;
  }
}
```

**Interoperability**

Because it is language neutral, Protobuf encoding means Data Grid can handle requests from client applications written in Java, C++, C#, Python, Go, and more.

Protobuf also enables clients on different remote endpoints, Hot Rod or REST, to operate on the same data. Because it uses the REST API, you can access and work with Protobuf-encoded caches through Data Grid Console.

**NOTE**

You cannot use Data Grid Console with any binary encoding other than \texttt{application/x-protostream}.

You should always use Protobuf cache encoding with the \texttt{application/x-protostream} media type for integration with any Red Hat technology because it allows communication between applications and services.

**Queries**

Data Grid needs a structured representation of data in caches for fast and reliable queries. To search caches with the Ickle query language, you register Protobuf schema that describe your objects.

**Custom types**

Data Grid includes an implementation of the ProtoStream API with native support for frequently used types, including \texttt{String} and \texttt{Integer}. If you want to store custom types in your caches, use ProtoStream marshalling to generate and register serialization contexts with Data Grid so that it can marshall your objects.

**Additional resources**

- Data Grid ProtoStream API
- developers.google.com/protocol-buffers
1.2.1. Encoding caches as ProtoStream

Configure Data Grid to use the ProtoStream library to store cache entries as Protocol Buffers (Protobuf).

Procedure

- Specify the application/x-protostream media type for keys and values.

Declarative

```xml
<distributed-cache name="mycache">
  <encoding>
    <key media-type="application/x-protostream"/>
    <value media-type="application/x-protostream"/>
  </encoding>
</distributed-cache>
```

Programmatic

```java
//Create cache configuration that encodes keys and values as ProtoStream
ConfigurationBuilder builder = new ConfigurationBuilder();
builder.clustering().cacheMode(CacheMode.DIST_SYNC)
  .encoding().key().mediaType("application/x-protostream")
  .encoding().value().mediaType("application/x-protostream");
```

Alternatively you can use the same encoding for keys and values:

Declarative

```xml
<encoding media-type="application/x-protostream"/>
```

Programmatic

```java
.encoding().mediaType("application/x-protostream");
```

Additional resources

- Data Grid Schema Reference
- org.infinispan.configuration.cache.ConfigurationBuilder

1.3. TEXT-BASED CACHE ENCODING

Text-based encoding is human-readable content such as plain text. The classic "Hello World" example entry could be stored in a cache as follows:

```text
key=hello
value=world
```
If you encode caches with the **text/plain** media type, Data Grid can convert to and from the following media types:

- application/xml
- application/json
- application/x-protostream
- application/x-jboss-marshalling
- application/x-java-serialized

The following example configuration encodes keys and values with the **text/plain; charset=UTF-8** media type:

```xml
<distributed-cache name="mycache">
  <encoding>
    <key media-type="text/plain; charset=UTF-8"/>
    <value media-type="text/plain; charset=UTF-8"/>
  </encoding>
</distributed-cache>
```

### 1.3.1. Clients and text-based encoding

If you configure encoding to store keys and values with a text-based media type, then you also need to configure clients to operate on those caches.

**Hot Rod clients**

Data Grid uses the ProtoStream library to handle `String` and `byte[]` types natively. If you configure cache encoding with the **text/plain** media type, Hot Rod clients might not necessarily require any marshaller configuration to perform cache operations.

For other text-based media types, such as JSON or XML, Hot Rod clients can use the `org.infinispan.commons.marshall.UTF8StringMarshaller` marshaller that converts to and from the **text/plain** media type.

**REST clients**

REST clients must include the media type for caches in the request headers.

For example if you configure cache encoding as **text/plain; charset=UTF-8** then REST clients should send the following headers:

- **Accept: text/plain; charset=UTF-8** for read operations.
- **Content-Type: text/plain; charset=UTF-8** or **Key-Content-Type: text/plain; charset=UTF-8** for write operations.

### Additional resources

- `org.infinispan.commons.marshall.UTF8StringMarshaller`

### 1.4. MARSHALLED JAVA OBJECTS
Data Grid stores marshalled Java objects in caches as byte arrays. For example, the following is a simple representation of a Person object stored as a value in memory:

```
value=[61 6c 61 6e 0a 70 61 72 74 72 69 64 67 65]
```

To store marshalled objects in caches, you should use the ProtoStream marshaller unless a strict requirement exists. For example, when migrating client applications from older versions of Data Grid, you might need to temporarily use JBoss marshalling with your Hot Rod Java clients.

Data Grid stores marshalled Java objects as byte arrays with the following media types:

- `application/x-protostream`
- `application/x-jboss-marshalling`
- `application/x-java-serialized-object`

**NOTE**

When storing unmarshalled Java objects, Data Grid uses the object implementation of `equals()` and `hashCode()`. When storing marshalled objects, the marshalled bytes are compared for equality and hashed instead.

### 1.4.1. Clients and marshalled objects

When you configure Hot Rod Java clients to use a marshaller, you must configure your cache with the encoding for that marshaller.

Each marshaller uses a different media type to produce `byte[]` content that the client can transmit to Data Grid Server. When reading from the server, the client marshaller performs the opposite operation, using the media type to produce data from `byte[]` content.

Your cache encoding must be compatible with the Hot Rod client marshaller. For example, if you configure a cache encoding as `application/x-protostream`, you can use the ProtoStream marshaller with your clients to operate on that cache. However if the client marshaller uses an encoding that Data Grid cannot convert to and from `application/x-protostream`, Data Grid throws an error message.

If you use `JavaSerializationMarshaller` or `GenericJBossMarshaller` you should encode caches with the `application/x-java-serialized-object` or `application/x-jboss-marshalling` media type, respectively.

**ProtoStream to JSON conversion**

Data Grid converts keys and values encoded with the `application/x-protostream` media type to `application/json`.

This allows REST clients to include the JSON media type in request headers and perform operations on caches that use ProtoStream encoding:

- **Accept:** `application/json` for read operations.
- **Content-Type:** `application/json` for write operations.

### 1.5. PLAIN OLD JAVA OBJECTS (POJO)

For best performance, Data Grid recommends storing unmarshalled POJOs in embedded caches only. However, you can configure keys and values with the following media type:
1.5.1. Clients and POJOs

Even though Data Grid does not recommend doing so, clients can operate on caches that store unmarshalled POJOs with the application/x-java-object media type.

Hot Rod clients
Hot Rod client marshellers must be available to Data Grid Server so it can deserialize your Java objects. By default, the ProtoStream and Java Serialization marshellers are available on the server.

REST clients
REST clients must use either JSON or XML for keys and values so Data Grid can convert to and from POJOs.

**NOTE**
Data Grid requires you to add Java classes to the deserialization allowlist to convert XML to and from POJOs.

1.6. ADDING JARS TO DATA GRID SERVER INSTALLATIONS

Make custom JAR files available to Data Grid Server by adding them to the classpath.

**IMPORTANT**
- Data Grid loads JAR files during startup only. You should bring all nodes in the cluster down gracefully and make any JAR files available to each node before bringing the cluster back up.
- You should add custom JAR files to the $RHDG_HOME/server/lib directory only. The $RHDG_HOME/lib directory is reserved for Data Grid JAR files.

**Procedure**

1. Stop Data Grid Server if it is running.

2. Add JAR files to the server/lib directory, for example:

```
server
    └── lib
        └── UserObjects.jar
```
CHAPTER 2. MARSHALLING CUSTOM OBJECTS WITH PROTOSTREAM

Marshalling is a process that converts Java objects into a binary format that can be transferred across the network or stored to disk. The reverse process, unmarshalling, transforms data from a binary format back into Java objects.

Data Grid performs marshalling and unmarshalling to:

- Send data to other Data Grid nodes in a cluster.
- Store data in persistent cache stores.
- Transmit objects between clients and remote caches.
- Store objects in native memory outside the JVM heap.
- Store objects in JVM heap memory when the cache encoding is not application/x-java-object.

When storing custom objects in Data Grid caches, you should use Protobuf-based marshalling with the ProtoStream marshaller.

2.1. PROTOSTREAM MARSHALLING

Data Grid provides the ProtoStream API so you can marshall Java objects as Protocol Buffers (Protobuf).

ProtoStream natively supports many different Java data types, which means you do not need to configure ProtoStream marshalling for those types. For custom or user types, you need to provide some information so that Data Grid can marshall those objects to and from your caches.

SerializationContext
A repository that contains Protobuf type definitions, loaded from Protobuf schemas (.proto files), and the accompanying marshallers.

SerializationContextInitializer
An interface that initializes a SerializationContext.

Additional resources
- org.infinispan.protostream.SerializationContext
- org.infinispan.protostream.SerializationContextInitializer

2.1.1. ProtoStream types

Data Grid uses a ProtoStream library that can handle the following types for keys and values, as well as the unboxed equivalents in the case of primitive types:

- byte[]
- Byte
- String
- Integer
- Long
- Double
- Float
- Boolean
- Short
- Character
- java.util.Date
- java.time.Instant

Additional type collections
The ProtoStream library includes several adapter classes for common Java types, for example:

- java.math.BigDecimal
- java.math.BigInteger
- java.util.UUID

Data Grid provides all adapter classes for some common JDK classes in the protostream-types artifact, which is included in the infinispan-core and infinispan-client-hotrod dependencies. You do not need any configuration to store adapter classes as keys or values.

However, if you want to use adapter classes as marshallable fields in ProtoStream-annotated POJOS, you can do so in the following ways:

- Specify the CommonTypesSchema and CommonContainerTypesSchema classes with the dependsOn element of the AutoProtoSchemaBuilder annotation.

```java
@AutoProtoSchemaBuilder(dependsOn = {org.infinispan.protostream.types.java.CommonTypes,
                                     org.infinispan.protostream.types.java.CommonContainerTypes},
                        schemaFileName = "library.proto",
                        schemaFilePath = "proto",
                        schemaPackageName = "example")
public interface LibraryInitializer extends SerializationContextInitializer {
}
```

- Specify the required adapter classes with the includeClasses element of the AutoProtoSchemaBuilder annotation.

```java
@AutoProtoSchemaBuilder(includeClasses = { Author.class, Book.class, UUIDAdaptor.class,
                                           java.math.BigInteger },
                        schemaFileName = "library.proto",
                        schemaFilePath = "proto",
                        schemaPackageName = "library")
public interface LibraryInitializer extends SerializationContextInitializer {
}
```

Additional resources
- Protocol Buffers
2.1.2. ProtoStream annotations

The ProtoStream API includes annotations that you can add to Java applications to define Protobuf schemas, which provide a structured format for your objects.

This topic provides additional details about ProtoStream annotations. You should refer to the documentation in the org.infinispan.protostream.annotations package for complete information.

**ProtoField**

@ProtoField defines a Protobuf message field.

This annotation is required and applies to fields as well as getter and setter methods. A class must have at least one field annotated with @ProtoField before Data Grid can marshall it as Protobuf.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Value</th>
<th>Optional or required</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>number</td>
<td>Integer</td>
<td>Required</td>
<td>Tag numbers must be unique within the class.</td>
</tr>
<tr>
<td>type</td>
<td>Type</td>
<td>Optional</td>
<td>Declares the Protobuf type of the field. If you do not specify a type, it is inferred from the Java property.</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td>You can use the @ProtoField(type) element to change the Protobuf type, similarly to changing Java int to fixed32. Any incompatible declarations for the Java property cause compiler errors.</td>
</tr>
<tr>
<td>collectionImplementation</td>
<td>Class</td>
<td>Optional</td>
<td>Indicates the actual collection type if the property type is an interface or abstract class.</td>
</tr>
</tbody>
</table>
### javaType
- **Value**: Class
- **Optional or required**: Optional
- **Description**: Indicates the actual Java type if the property type is an abstract class or interface. The value must be an instantiable Java type assignable to the property type.

If you declare a type with the `javaType` parameter, then all user code must adhere to that type. The generated marshaller for the entry uses that implementation if it is unmarshalled. If the local client uses a different implementation than declared it causes ClassCastException.

### name
- **Value**: String
- **Optional or required**: Optional
- **Description**: Specifies a name for the Protobuf schema.

### defaultValue
- **Value**: String
- **Optional or required**: Optional
- **Description**: Specifies the default value for fields if they are not available when reading from the cache. The value must follow the correct syntax for the Java field type.

---

**ProtoFactory**

`@ProtoFactory` marks a single constructor or static factory method for creating instances of the message class.

You can use this annotation to support immutable message classes. All fields annotated with `@ProtoField` must be included in the parameters.

- Field names and parameters of the `@ProtoFactory` constructor or method must match the corresponding Protobuf message, however, the order is not important.

- If you do not add a `@ProtoFactory` annotated constructor to a class, that class must have a default no-argument constructor, otherwise errors occur during compilation.

**AutoProtoSchemaBuilder**

`@AutoProtoSchemaBuilder` generates an implementation of a class or interface that extends `SerializationContextInitializer`.

---
If active, the ProtoStream processor generates the implementation at compile time in the same package with the `impl` suffix or a name that you specify with the `className` parameter.

The `includeClasses` or `basePackages` parameters reference classes that the ProtoStream processor should scan and include in the Protobuf schema and marshaller. If you do not set either of these parameters, the ProtoStream processor scans the entire source path, which can lead to unexpected results and is not recommended. You can also use the `excludeClasses` parameter with the `basePackages` parameter to exclude classes.

The `schemaFileName` and `schemaPackageName` parameters register the generated Protobuf schema under this name. If you do not set these parameters, the annotated simple class name is used with the unnamed, or default, package. Schema names must end with the `.proto` file extension. You can also use the `marshellersOnly` to generate marshellers only and suppress the Protobuf schema generation.

The ProtoStream process automatically generates `META-INF/services` service metadata files, which you can use so that Data Grid Server automatically picks up the JAR to register the Protobuf schema.

The `dependsOn` parameter lists annotated classes that implement `SerializedContextInitializer` to execute first. If the class does not implement `SerializedContextInitializer` or is not annotated with `AutoProtoSchemaBuilder`, a compile time error occurs.

ProtoAdapter
@ProtoAdapter is a marshalling adapter for a class or enum that you cannot annotate directly.

If you use this annotation for:

- Classes, the annotated class must have one `@ProtoFactory` annotated factory method for the marshalled class and annotated accessor methods for each field. These methods can be instance or static methods and their first argument must be the marshalled class.

- Enums, an identically named enum value must exist in the target enum.

ProtoDoc and ProtoDocs
@ProtoDoc and @ProtoDocs are human-readable text that document message types, enums, or fields for the generated schema.

You use these annotation to configure indexing for Ickle queries.

ProtoName
@ProtoName is an optional annotation that specifies the Protobuf message or enum type name and replaces the `@ProtoMessage` annotation.

ProtoEnumValue
@ProtoEnumValue defines a Protobuf enum value. You can apply this annotation to members of a Java enum only.

ProtoReserved and ProtoReservedStatements
@ProtoReserved and @ProtoReservedStatements add `reserved` statements to generated messages or enum definitions to prevent future usage of numbers, ranges, and names.

ProtoTypeId
@ProtoTypeId optionally specifies a globally unique numeric type identifier for a Protobuf message or enum type.
NOTE

You should not add this annotation to classes because Data Grid uses it internally and identifiers can change without notice.

ProtoUnknownFieldSet
@ProtoUnknownFieldSet optionally indicates the field, or JavaBean property of type  {@link org.infinispan.protostream.UnknownFieldSet}, which stores any unknown fields.

NOTE

Data Grid does not recommend using this annotation because it is no longer supported by Google and is likely to be removed in future.

Additional resources

- org.infinispan.protostream.annotations
- Querying values in caches
- Protocol Buffers Language Guide - Reserved Fields
- Protocol Buffers Language Guide - Reserved Values

2.2. CREATING SERIALIZATION CONTEXT INITIALIZERS

A serialization context initializer lets you register the following with Data Grid:

- Protobuf schemas that describe user types.
- Marshellers that provide serialization and deserialization capabilities.

From a high level, you should do the following to create a serialization context initializer:

1. Add ProtoStream annotations to your Java classes.

2. Use the ProtoStream processor that Data Grid provides to compile your SerializationContextInitializer implementation.

NOTE

The org.infinispan.protostream.MessageMarshaller interface is deprecated and planned for removal in a future version of ProtoStream. You should ignore any code examples or documentation that show how to use MessageMarshaller until it is completely removed.

2.2.1. Adding the ProtoStream processor

Data Grid provides a ProtoStream processor artifact that processes Java annotations in your classes at compile time to generate Protobuf schemas, accompanying marshellers, and a concrete implementation of the SerializationContextInitializer interface.

Procedure
Add the `protostream-processor` dependency to your `pom.xml` with the `provided` scope.

**NOTE**

This dependency is required at compile-time only so you should use the `provided` scope or mark it as optional. You should also ensure the `protostream-processor` is not propagated as a transitive dependency.

```xml
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>org.infinispan</groupId>
      <artifactId>infinispan-bom</artifactId>
      <version>${version.infinispan}</version>
      <type>pom</type>
    </dependency>
    <dependency>
      <groupId>org.infinispan.protostream</groupId>
      <artifactId>protostream-processor</artifactId>
      <scope>provided</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

### 2.2.2. Adding ProtoStream annotations to Java classes

Declare ProtoStream metadata by adding annotations to a Java class and its members. Data Grid then uses the ProtoStream processor to generate Protobuf schema and related marshellers from those annotations.

**Procedure**

1. Annotate the Java fields that you want to marshall with `@ProtoField`, either directly on the field or on the getter or setter method. Any non-annotated fields in your Java class are transient. For example, you have a Java class with 15 fields and annotate five of them. The resulting schema contains only those five fields and only those five fields are marshalled when storing a class instance in Data Grid.

2. Use `@ProtoFactory` to annotate constructors for immutable objects. The annotated constructors must initialize all fields annotated with `@ProtoField`.

3. Annotate members of any Java enum with `@ProtoEnumValue`.

The following `Author.java` and `Book.java` examples show Java classes annotated with `@ProtoField` and `@ProtoFactory`:

**Author.java**

```java
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
```
The following `Language.java` example shows a Java enum annotated with `@ProtoEnumValue` along with the corresponding Protobuf schema:
Language.java

```java
import org.infinispan.protostream.annotations.ProtoEnumValue;

public enum Language {
    @ProtoEnumValue(number = 0, name = "EN") ENGLISH,
    @ProtoEnumValue(number = 1, name = "DE") GERMAN,
    @ProtoEnumValue(number = 2, name = "IT") ITALIAN,
    @ProtoEnumValue(number = 3, name = "ES") SPANISH,
    @ProtoEnumValue(number = 4, name = "FR") FRENCH;
}
```

Language.proto

```java
enum Language {
    EN = 0;
    DE = 1;
    IT = 2;
    ES = 3;
    FR = 4;
}
```

Additional resources

- [org.infinispan.protostream.annotations.ProtoField](#)
- [org.infinispan.protostream.annotations.ProtoFactory](#)

### 2.2.3. Creating ProtoStream adapter classes

ProtoStream provides a `@ProtoAdapter` annotation that you can use to marshall external, third-party Java object classes that you cannot annotate directly.

**Procedure**

1. Create an `Adapter` class and add the `@ProtoAdapter` annotation, as in the following example:

```java
import java.util.UUID;
import org.infinispan.protostream.annotations.ProtoAdapter;
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;
import org.infinispan.protostream.descriptors.Type;
```
2.2.4. Generating serialization context initializers

After you add the ProtoStream processor and annotate your Java classes, you can add the `@AutoProtoSchemaBuilder` annotation to an interface so that Data Grid generates the Protobuf schema, accompanying marshallers, and a concrete implementation of the `SerializationContextInitializer`.

**NOTE**

By default, generated implementation names are the annotated class name with an "Impl" suffix.

**Procedure**

1. Define an interface that extends `GeneratedSchema` or its super interface, `SerializationContextInitializer`.

   **NOTE**

   The `GeneratedSchema` interface includes a method to access the Protobuf schema whereas the `SerializationContextInitializer` interface supports only registration methods.

2. Annotate the interface with `@AutoProtoSchemaBuilder`.

3. Ensure that `includeClasses` parameter includes all classes for the generated `SerializationContextInitializer` implementation.

4. Specify a name for the generated `.proto` schema with the `schemaFileName` parameter.

5. Set a path under `target/classes` where schema files are generated with the `schemaFilePath` parameter.
6. Specify a package name for the generated .proto schema with the schemaPackageName parameter.

The following example shows a GeneratedSchema interface annotated with @AutoProtoSchemaBuilder:

```java
@AutoProtoSchemaBuilder(
    includeClasses = {
        Book.class,
        Author.class,
        UUIDAdapter.class,
        Language.class
    },
    schemaFileName = "library.proto",
    schemaFilePath = "proto/",
    schemaPackageName = "book_sample")
interface LibraryInitializer extends GeneratedSchema {
}
```

Next steps

If you use embedded caches, Data Grid automatically registers your SerializationContextInitializer implementation.

If you use remote caches, you must register your SerializationContextInitializer implementation with Data Grid Server.

Additional resources

- org.infinispan.protostream.annotations.AutoProtoSchemaBuilder

2.2.5. Registering serialization context initializers

For embedded caches, Data Grid automatically registers serialization contexts and marshallers in your annotated SerializationContextInitializer implementation using the java.util.ServiceLoader.

If you prefer, you can disable automatic registration of SerializationContextInitializer implementations and then register them manually.

**IMPORTANT**

If you manually register one SerializationContextInitializer implementation, it disables automatic registration. You must then manually register all other implementations.

**Procedure**

1. Set a value of false for the AutoProtoSchemaBuilder.service annotation.

```java
@AutoProtoSchemaBuilder(
    includeClasses = SomeClass.class,
    ...
    service = false
)
```
2. Manually register `SerializationContextInitializer` implementations either programmatically or declaratively, as in the following examples:

**Declarative**

```xml
<serialization>
  <context-initializer class="org.infinispan.example.LibraryInitializerImpl"/>
  <context-initializer class="org.infinispan.example.another.SCIImpl"/>
</serialization>
```

**Programmatic**

```java
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
  .addContextInitializers(new LibraryInitializerImpl(), new SCIImpl());
```

### 2.2.6. Registering Protobuf schemas with Data Grid Server

You must register Protobuf schemas with Data Grid Server if you want to perform Ickle queries or convert from `application/x-protostream` to other media types such as `application/json`.

**Prerequisites**

- Generate a `SerializationContextInitializer` implementation with the ProtoStream processor. After the build completes, you can find generated Protobuf schema in the `target/<schemaFilePath>/` directory.

**Procedure**

Do one of the following to register your generated Protobuf schema with Data Grid Server:

- Add the generated Protobuf schema to the dedicated `__protobuf_metadata` cache with the Data Grid command line interface (CLI), REST interface, or `ProtobufMetadataManager` JMX MBean. This method ensures that Data Grid automatically distributes your schema across the cluster.

  **NOTE**

  If you enable security authorization for caches, users must have the `CREATE` permission to add schemas to the `__protobuf_metadata` cache. Assign users the `deployer` role at minimum if you use default authorization settings.

- Use the generated `SerializationContextInitializer` implementation with a Hot Rod client to register the Protobuf schema, as in the following example:

```java
/**
 * Register generated Protobuf schema with Data Grid Server.
 * This requires the RemoteCacheManager to be initialized.
 *
 * @param initializer The serialization context initializer for the schema.
 */
private void registerSchemas(SerializationContextInitializer initializer) {
  // Store schemas in the ‘__protobuf_metadata’ cache to register them.
```
Add a JAR file with the SerializationContextInitializer implementation and custom classes to the $RHDG_HOME/server/lib directory. When you do this, Data Grid Server registers your Protobuf schema at startup. However, you must add the archive to each server installation because the schema are not saved in the __protobuf_metadata cache or automatically distributed across the cluster.

NOTE
You must do this if you require Data Grid Server to perform any application/x-protostream to application/x-java-object conversions, in which case you must also add any JAR files for your POJOs.

Next steps
Register the SerializationContextInitializer with your Hot Rod clients, as in the following example:

```java
@ConfigurationBuilder
public class RemoteConfiguration {
    @Override
default void setRemoteBuilder(ConfigurationBuilder remoteBuilder) {
        remoteBuilder.addServer().host(host).port(Integer.parseInt(port));
        // Add your generated SerializationContextInitializer implementation.
        LibraryInitializer initializer = new LibraryInitializerImpl();
        remoteBuilder.addContextInitializer(initializer);
    }
}
```

Additional resources

- CLI: Registering Protobuf schemas
- REST API: Creating Protobuf schemas

2.2.7. Manual serialization context initializer implementations

IMPORTANT
Data Grid strongly recommends against manually implementing the SerializationContextInitializer or GeneratedSchema interfaces.
It is possible to manually implement `SerializationContextInitializer` or `GeneratedSchema` interfaces using `ProtobufTagMarshaller` and `RawProtobufMarshaller` annotations.

However, manual implementations require a lot of tedious overhead and are prone to error. Implementations that you generate with the `protostream-processor` artifact are a much more efficient and reliable way to configure ProtoStream marshalling.
CHAPTER 3. USING ALTERNATIVE AND CUSTOM MARSHALLER IMPLEMENTATIONS

Data Grid recommends you use Protobuf-based marshalling with the ProtoStream marshaller so you can take advantage of Ickle queries and use the Data Grid CLI and Console. However, if required, you can use alternative marshellers or a custom marshaller implementation.

3.1. ALLOWING DESERIALIZATION OF JAVA CLASSES

For security reasons Data Grid does not allow deserialization of arbitrary Java classes. If you use JavaSerializationMarshaller or GenericJBossMarshaller, you must add your Java classes to a deserialization allow list.

NOTE

The deserialization allow list applies to the Cache Manager so your Java classes can be deserialized by all caches.

Procedure

- Add Java classes to the deserialization allow list in the Data Grid configuration or in system properties.

Declarative

```xml
<cache-container>
  <serialization version="1.0">
    <marshaller>org.infinispan.marshall.TestObjectStreamMarshaller</marshaller>
    <allow-list>
      <class>org.infinispan.test.data.Person</class>
      <regex>org.infinispan.test.data.*</regex>
    </allow-list>
  </serialization>
</cache-container>
```

System properties

```bash
-Dinfinispan.deserialization.allowlist.classes=java.time.Instant,com.myclass.Entity
-Dinfinispan.deserialization.allowlist.regexps=.*
```

3.2. USING JBOSS MARSHALLING

JBoss Marshalling is a serialization-based marshalling library and was the default marshaller in previous Data Grid versions but is now deprecated.
NOTE

JBoss Marshalling is deprecated. You should use it only as a temporary measure while migrating your applications from an older version of Data Grid.

Procedure

1. Add the `infinispan-jboss-marshalling` dependency to your classpath.
2. Configure Data Grid to use the `GenericJBossMarshaller`.
3. Add your Java classes to the deserialization allowlist.

Declarative

```xml
<serialization marshaller="org.infinispan.jboss.marshalling.commons.GenericJBossMarshaller">
  <allow-list>
    <class>org.infinispan.concrete.SomeClass</class>
    <regex>org.infinispan.example.*</regex>
  </allow-list>
</serialization>
```

Programmatic

```java
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization().marshaller(new GenericJBossMarshaller())
  .allowList()
  .addRegexp("org.infinispan.example.", "org.infinispan.concrete.SomeClass");
```

Additional resources

- AdvancedExternalizer

3.3. USING JAVA SERIALIZATION

You can use Java serialization with Data Grid to marshall objects that implement the Java `Serializable` interface.

TIP

Java serialization offers worse performance than ProtoStream marshalling. You should use Java serialization only if there is a strict requirement to do so.

Procedure

1. Configure Data Grid to use `JavaSerializationMarshaller`.
2. Add your Java classes to the deserialization allowlist.

Declarative
3.4. USING CUSTOM MARSHALLERS

Data Grid provides a **Marshaller** interface that you can implement for custom marshellers.

**TIP**

Custom marshaller implementations can access a configured access list via the `initialize()` method, which is called during startup.

**Procedure**

1. Implement the **Marshaller** interface.
2. Configure Data Grid to use your marshaller.
3. Add your Java classes to the deserialization allowlist.

**Declarative**

```
<serialization marshaller="org.infinispan.example.marshall.CustomMarshaller">
  <allow-list>
    <class>org.infinispan.concrete.SomeClass</class>
    <regex>org.infinispan.example.*</regex>
  </allow-list>
</serialization>
```

**Programmatic**

```java
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization().marshaller(new JavaSerializationMarshaller())
  .allowList()
  .addRegexp("org.infinispan.example.", "org.infinispan.concrete.SomeClass");
```

**Additional resources**

- **Serializable**
- `org.infinispan.commons.marshall.JavaSerializationMarshaller`
.marshaller(new org.infinispan.example.marshall.CustomMarshaller())
.allowList().addRegexp("org.infinispan.example.**");

Additional resources

- org.infinispan.commons.marshall.Marshaller
CHAPTER 4. DATA CONVERSION

Data Grid uses transcoders to convert data between various encodings that are identified by media
types.

4.1. HOT ROD DATAFORMAT API

Read and write operations on remote caches via the Hot Rod endpoint use the client marshaller by
default. Hot Rod provides a DataFormat API for Java clients that you can use to perform cache
operations with different media type encodings and/or marshaller.

Different marshallers for key and values

You can override marshallers for keys and values at run time.

For example, to bypass all serialization in the Hot Rod client and read the byte[] array stored in the
remote cache:

```java
// Existing RemoteCache instance
RemoteCache<String, Pojo> remoteCache = ...

// IdentityMarshaller is a no-op marshaller
DataFormat rawKeyAndValues =
    DataFormat.builder()
        .keyMarshaller(IdentityMarshaller.INSTANCE)
        .valueMarshaller(IdentityMarshaller.INSTANCE)
        .build();

// Creates a new instance of RemoteCache with the supplied DataFormat
RemoteCache<byte[], byte[]> rawResultsCache =
    remoteCache.withDataFormat(rawKeyAndValues);
```

**IMPORTANT**

Using different marshallers and data formats for keys with keyMarshaller() and
keyType() methods can interfere with client intelligence routing mechanisms, causing
extra network hops within the Data Grid cluster. If performance is critical, you should use
the same encoding for keys on the client and on the server.

Reading data in different encodings

Request and send data in different encodings specified by a
org.infinispan.commons.dataconversion.MediaType as follows:

```java
// Existing remote cache using ProtostreamMarshaller
RemoteCache<String, Pojo> protobufCache = ...

// Request values returned as JSON
// Use the UTF8StringMarshaller to convert UTF-8 to String
DataFormat jsonString =
    DataFormat.builder()
        .valueType(MediaType.APPLICATION_JSON)
        .valueMarshaller(new UTF8StringMarshaller()));
```
Using custom value marshallers

You can use custom marshallers for values, as in the following example that returns values as `org.codehaus.jackson.JsonNode` objects.

In this example, Data Grid Server handles the data conversion and throws an exception if it does not support the specified media type.

```java
DataFormat jsonNode = DataFormat.builder()
    .valueType(MediaType.APPLICATION_JSON)
    .valueMarshaller(new CustomJacksonMarshaller().build());

RemoteCache<String, JsonNode> jsonNodeCache = remoteCache.withDataFormat(jsonNode);
```

Returning values as XML

The following code snippet returns values as XML:

```java
Object xmlValue = remoteCache
    .withDataFormat(DataFormat.builder()
        .valueType(MediaType.APPLICATION_XML)
        .valueMarshaller(new UTF8StringMarshaller())
        .build())
    .get(key);
```

For example, the preceding `get(key)` call returns values such as:

```xml
<?xml version="1.0"?><string>Hello!</string>
```

Reference

`org.infinispan.client.hotrod.DataFormat`

4.2. CONVERTING DATA ON DEMAND WITH EMBEDDED CACHES

Embedded caches have a default request encoding of `application/x-java-object` and a storage encoding that corresponds to the media type that you configure for the cache. This means that Data Grid marshalls POJOs from the application to the storage encoding for the cache and then returns POJOs back to the application. In some complex scenarios you can use the `AdvancedCache` API to change the default conversion to and from POJOs to other encodings.

The following example uses the `withMediaType()` method to return values as `application/json` on demand.

Advanced cache with MediaType

```java
DefaultCacheManager cacheManager = new DefaultCacheManager();
```
// Encode keys and values as Protobuf
ConfigurationBuilder cfg = new ConfigurationBuilder();
cfg.encoding().key().mediaType("application/x-protostream");
cfg.encoding().value().mediaType("application/x-protostream");

cacheManager.defineConfiguration("mycache", cfg.build());

Cache<Integer, Person> cache = cacheManager.getCache("mycache");

cache.put(1, new Person("John", "Doe"));

// Use Protobuf for keys and JSON for values
Cache<Integer, byte[]> jsonValuesCache = (Cache<Integer, byte[]>)
cache.getAdvancedCache().withMediaType("application/x-protostream", "application/json");

byte[] json = jsonValuesCache.get(1);

Value returned in JSON format

{
  "_type": "org.infinispan.sample.Person",
  "name": "John",
  "surname": "Doe"
}

Additional resources

- org.infinispan.AdvancedCache