Marshalling and Encoding Data

Table of Contents

Infinispan caches can store keys and values in different encodings. This document describes how Infinispan encodes data for remote and embedded caches and explains how to use various media types with your applications.

Chapter 1. Configuring cache encoding

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

1.1. Cache encoding

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

Remote caches

Infinispan 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 Infinispan Server. To handle multiple clients making read and write requests with different media types, Infinispan 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, Infinispan 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

Infinispan 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.

Infinispan recommends always configuring cache encoding with the `application/x-protostream` media type if you want to use multiple clients, such as Infinispan 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

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

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

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 Infinispan 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 Infinispan is marshalling the object.

Additional resources

- Infinispan 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.

Infinispan uses a ProtoStream library to encode caches as Protobuf with the `application/x-protostream` media type.

The following example shows a Protobuf message that describes a `Person` object:

```
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 Infinispan 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 Infinispan Console.

You cannot use Infinispan Console with any binary encoding other than `application/x-protostream`.

Queries

Infinispan 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

Infinispan includes an implementation of the ProtoStream API with native support for frequently

used types, including `String` and `Integer`. If you want to store custom types in your caches, use ProtoStream marshalling to generate and register serialization contexts with Infinispan so that it can marshall your objects.

Additional resources

- Infinispan ProtoStream API
- developers.google.com/protocol-buffers

1.2.1. Encoding caches as ProtoStream

Configure Infinispan 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

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

Programmatic

```
//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

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

Programmatic

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

Additional resources

- Infinispan 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:

```
key=hello
value=world
```

If you encode caches with the `text/plain` media type, Infinispan 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:

```
<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

Infinispan 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

Infinispan 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 Infinispan, you might need to temporarily use JBoss marshalling with your Hot Rod Java clients.

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

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

When storing unmarshalled Java objects, Infinispan 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 Infinispan 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 Infinispan cannot convert to and from `application/x-protostream`, Infinispan 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

Infinispan 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, Infinispan recommends storing unmarshalled POJOs in embedded caches only. However, you can configure keys and values with the following media type:

- `application/x-java-object`

1.5.1. Clients and POJOs

Even though Infinispan 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 marshallers must be available to Infinispan Server so it can deserialize your Java objects. By default, the ProtoStream and Java Serialization marshallers are available on the server.

REST clients

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

Infinispan requires you to add Java classes to the deserialization allowlist to convert XML to and from POJOs.

1.6. Adding JARs to Infinispan Server installations

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

- Infinispan 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 `$ISPN_HOME/server/lib` directory only.

  The `$ISPN_HOME/lib` directory is reserved for Infinispan JAR files.

Procedure

1. Stop Infinispan Server if it is running.
2. Add JAR files to the `server/lib` directory, for example:

```
├──── server
│     └──── lib
│           └──── UserObjects.jar
```

1.7. Configuring cache encoding for Memcached clients

Infinispan Server disables the Memcached endpoint by default. If you enable the Memcached endpoint, you should configure your Memcached cache with a suitable encoding.

The Memcached endpoint does not support authentication. For security purposes you should use dedicated caches for Memcached clients. You should not use REST or Hot Rod clients to interact on the same data set as Memcached clients.

Procedure

1. Configure cache encoding for keys and values as appropriate.
2. Specify any appropriate media type for values.

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

Encoding

The Memcached endpoint includes a `client-encoding` attribute that converts the encoding of values.

For example, as in the preceding configuration example, you store values encoded as Protobuf. If you want Memcached clients to read and write values as JSON, you can use the following

configuration:

```
<memcached-connector cache="memcachedCache" client-encoding="application/json">
```

Additional resources

- Infinispan Memcached Client Guide

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.

Infinispan performs marshalling and unmarshalling to:

- Send data to other Infinispan 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 Infinispan caches, you should use Protobuf-based marshalling with the ProtoStream marshaller.

2.1. ProtoStream marshalling

Infinispan 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 Infinispan 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

Infinispan 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`

Infinispan 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.

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

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

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

}
```

Additional resources

- Protocol Buffers

- Infinispan ProtoStream API
- Infinispan ProtoStream library

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 Infinispan can marshall it as Protobuf.

Parameter	Value	Optional or required	Description
`number`	Integer	Required	Tag numbers must be unique within the class.
`type`	Type	Optional	Declares the Protobuf type of the field. If you do not specify a type, it is inferred from the Java property. 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.
`collectionImplementation`	Class	Optional	Indicates the actual collection type if the property type is an interface or abstract class.

Parameter	Value	Optional or required	Description
`javaType`	Class	Optional	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 ClassCastExceptions.
`name`	String	Optional	Specifies a name for the Protobuf schema.
`defaultValue`	String	Optional	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 `marshallersOnly` to generate marshallers only and suppress the Protobuf schema generation.

The ProtoStream process automatically generates `META-INF/services` service metadata files, which you can use so that Infinispan 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.

You should not add this annotation to classes because Infinispan 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.

Infinispan 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 Infinispan:

- Protobuf schemas that describe user types.
- Marshallers 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 Infinispan provides to compile your `SerializationContextInitializer` implementation.

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

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

Procedure

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

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.

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

<dependencies>
  <dependency>
    <groupId>org.infinispan.protostream</groupId>
    <artifactId>protostream-processor</artifactId>
    <scope>provided</scope>
  </dependency>
</dependencies>
```

2.2.2. Adding ProtoStream annotations to Java classes

Declare ProtoStream metadata by adding annotations to a Java class and its members. Infinispan then uses the ProtoStream processor to generate Protobuf schema and related marshallers 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 Infinispan.

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

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

public class Author {
   @ProtoField(1)
   final String name;

   @ProtoField(2)
   final String surname;

   @ProtoFactory
   Author(String name, String surname) {
      this.name = name;
      this.surname = surname;
   }
   // public Getter methods omitted for brevity
}
```

Book.java

```
import org.infinispan.protostream.annotations.ProtoFactory;
import org.infinispan.protostream.annotations.ProtoField;

public class Book {
   @ProtoField(number = 1)
   public final UUID id;

   @ProtoField(number = 2)
   final String title;

   @ProtoField(number = 3)
   final String description;

   @ProtoField(number = 4, defaultValue = "0")
   final int publicationYear;

   @ProtoField(number = 5, collectionImplementation = ArrayList.class)
   final List<Author> authors;

   @ProtoField(number = 6)
   public Language language;

   @ProtoFactory
   Book(UUID id, String title, String description, int publicationYear, List<Author>
authors, Language language) {
      this.id = id;
      this.title = title;
      this.description = description;
      this.publicationYear = publicationYear;
      this.authors = authors;
      this.language = language;
   }
   // public Getter methods not included for brevity
}
```

The following `Language.java` example shows a Java enum annotated with `@ProtoEnumValue` along with the corresponding Protobuf schema:

Language.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

```
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:

```
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;

/**
 * Human readable UUID adapter for UUID marshalling
 */
@ProtoAdapter(UUID.class)
public class UUIDAdapter {

  @ProtoFactory
  UUID create(String stringUUID) {
    return UUID.fromString(stringUUID);
  }

  @ProtoField(1)
  String getStringUUID(UUID uuid) {
    return uuid.toString();
  }
}
```

Additional resources

- org.infinispan.protostream.annotations.ProtoAdapter

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 Infinispan generates the Protobuf schema, accompanying marshallers, and a concrete implementation of the `SerializationContextInitializer`.

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`.

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`:

```
@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, Infinispan automatically registers your `SerializationContextInitializer` implementation.

If you use remote caches, you must register your `SerializationContextInitializer` implementation with Infinispan Server.

Additional resources

- org.infinispan.protostream.annotations.AutoProtoSchemaBuilder

2.2.5. Registering serialization context initializers

For embedded caches, Infinispan 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.

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.

```
@AutoProtoSchemaBuilder(
      includeClasses = SomeClass.class,
      ...
      service = false
)
```

2. Manually register `SerializationContextInitializer` implementations either programmatically or declaratively, as in the following examples:

Declarative

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

Programmatic

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

2.2.6. Registering Protobuf schemas with Infinispan Server

You must register Protobuf schemas with Infinispan 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 Infinispan Server:

- Add the generated Protobuf schema to the dedicated `___protobuf_metadata` cache with the Infinispan command line interface (CLI), REST interface, or `ProtobufMetadataManager` JMX MBean.

  This method ensures that Infinispan automatically distributes your schema across the cluster.

  

  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:

```
/**
 * Register generated Protobuf schema with Infinispan 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.
  // Using ProtobufMetadataManagerConstants might require the query dependency.
  final RemoteCache<String, String> protoMetadataCache = remoteCacheManager
.getCache(ProtobufMetadataManagerConstants.PROTOBUF_METADATA_CACHE_NAME);
  // Add the generated schema to the cache.
  protoMetadataCache.put(initializer.getProtoFileName(), initializer.getProtoFile(
));

  // Ensure the registered Protobuf schemas do not contain errors.
  // Throw an exception if errors exist.
  String errors = protoMetadataCache.get(ProtobufMetadataManagerConstants
.ERRORS_KEY_SUFFIX);
  if (errors != null) {
    throw new IllegalStateException("Some Protobuf schema files contain errors: " +
errors + "\nSchema :\n" + initializer.getProtoFileName());
  }
}
```

- Add a JAR file with the `SerializationContextInitializer` implementation and custom classes to the `$ISPN_HOME/server/lib` directory.

  When you do this, Infinispan 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.

You must do this if you require Infinispan 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:

```
ConfigurationBuilder remoteBuilder = new ConfigurationBuilder();
remoteBuilder.addServer().host(host).port(Integer.parseInt(port));

// Add your generated SerializationContextInitializer implementation.
LibraryInitalizer initializer = new LibraryInitalizerImpl();
remoteBuilder.addContextInitializer(initializer);
```

Additional resources

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

2.2.7. Manual serialization context initializer implementations

Infinispan 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

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

3.1. Allowing deserialization of Java classes

For security reasons Infinispan 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.

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 Infinispan configuration or in system properties.

Declarative

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

System properties

```
// Specify a comma-separated list of fully qualified class names
-Dinfinispan.deserialization.allowlist.classes=java.time.Instant,com.myclass.Entity

// Specify a regular expression to match classes
-Dinfinispan.deserialization.allowlist.regexps=.*
```

3.2. Using JBoss Marshalling

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

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

Procedure

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

Declarative

```
<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

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

Additional resources

- AdvancedExternalizer

3.3. Using Java serialization

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

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 Infinispan to use `JavaSerializationMarshaller`.
2. Add your Java classes to the deserialization allowlist.

Declarative

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

Programmatic

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

Additional resources

- Serializable
- org.infinispan.commons.marshall.JavaSerializationMarshaller

3.4. Using custom marshallers

Infinispan provides a `Marshaller` interface that you can implement for custom marshallers.

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 Infinispan 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

```
GlobalConfigurationBuilder builder = new GlobalConfigurationBuilder();
builder.serialization()
      .marshaller(new org.infinispan.example.marshall.CustomMarshaller())
      .allowList().addRegexp("org.infinispan.example.*");
```

Additional resources

- org.infinispan.commons.marshall.Marshaller

Chapter 4. Data conversion

Infinispan 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 marshallers.

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:

```
// 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);
```

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 Infinispan 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:

```
// 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())
          .build();
RemoteCache<byte[], byte[]> rawResultsCache =
protobufCache.withDataFormat(jsonString);
```

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, Infinispan Server handles the data conversion and throws an exception if it does not support the specified media type.

```
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:

```
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 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 Infinispan 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

```
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