

Quarkus - Infinispan Client

Infinispan is an in memory data grid that allows running in a server outside of application processes. This extension provides functionality to allow the client that can connect to said server when running in Quarkus.

More information can be found about Infinispan at <https://infinispan.org> and the client/server at https://infinispan.org/docs/dev/user_guide/user_guide.html#client_server

Configuration

Once you have your Quarkus project configured you can add the `infinispan-client` extension to your project by running the following from the command line in your project base directory.

```
./mvnw quarkus:add-extension -Dextensions="infinispan-client"
```

This will add the following to your pom.xml

```
<dependency>
  <groupId>io.quarkus</groupId>
  <artifactId>quarkus-infinispan-client</artifactId>
</dependency>
```

The Infinispan client is configurable in the `application.properties` file that can be provided in the `src/main/resources` directory. These are the properties that can be configured in this file:

 Configuration property fixed at build time - All other configuration properties are overridable at runtime

Configuration property	Type	Default
 <code>quarkus.infinispan-client.near-cache-max-entries</code> Sets the bounded entry count for near cache. If this value is 0 or less near cache is disabled.	int	0
<code>quarkus.infinispan-client.server-list</code> Sets the host name/port to connect to. Each one is separated by a semicolon (eg. host1:11222;host2:11222).	string	
<code>quarkus.infinispan-client.client-intelligence</code> Sets client intelligence used by authentication	string	

<code>quarkus.infinispan-client.use-auth</code>		
Enables or disables authentication	string	
<code>quarkus.infinispan-client.auth-username</code>		
Sets user name used by authentication	string	
<code>quarkus.infinispan-client.auth-password</code>		
Sets password used by authentication	string	
<code>quarkus.infinispan-client.auth-realm</code>		
Sets realm used by authentication	string	
<code>quarkus.infinispan-client.auth-server-name</code>		
Sets server name used by authentication	string	
<code>quarkus.infinispan-client.auth-client-subject</code>		
Sets client subject used by authentication	string	
<code>quarkus.infinispan-client.auth-callback-handler</code>		
Sets callback handler used by authentication	string	
<code>quarkus.infinispan-client.sasl-mechanism</code>		
Sets SASL mechanism used by authentication	string	
<code>quarkus.infinispan-client.trust-store</code>		
Sets the trust store path	string	
<code>quarkus.infinispan-client.trust-store-password</code>		
Sets the trust store password	string	
<code>quarkus.infinispan-client.trust-store-type</code>		
Sets the trust store type	string	

It is also possible to configure a `hotrod-client.properties` as described in the Infinispan user guide. Note that the `hotrod-client.properties` values overwrite any matching property from the other configuration values (eg. near cache). This properties file is build time only and if it is changed, requires a full rebuild.

Serialization (Key Value types support)

By default the client will support keys and values of the following types: byte[], primitive wrappers (eg. Integer, Long, Double etc.), String, Date and Instant. User types require some additional steps that are detailed here. Let's say we have the following user classes:

Author.java

```
public class Author {
    private final String name;
    private final String surname;

    public Author(String name, String surname) {
        this.name = Objects.requireNonNull(name);
        this.surname = Objects.requireNonNull(surname);
    }
    // Getter/Setter>equals/hashCode/toString omitted
}
```

Book.java

```
public class Book {
    private final String title;
    private final String description;
    private final int publicationYear;
    private final Set<Author> authors;

    public Book(String title, String description, int
publicationYear, Set<Author> authors) {
        this.title = Objects.requireNonNull(title);
        this.description = Objects.requireNonNull(description);
        this.publicationYear = publicationYear;
        this.authors = Objects.requireNonNull(authors);
    }
    // Getter/Setter>equals/hashCode/toString omitted
}
```

Serialization of user types uses a library based on protobuf, called Protostream.

Annotation based Serialization

This can be done automatically by adding protostream annotations to your user classes. In addition a single Initializer annotated interface is required which controls how the supporting classes are generated.

Here is an example of how the preceding classes should be changed:

Author.java

```
@ProtoFactory
public Author(String name, String surname) {
    this.name = Objects.requireNonNull(name);
    this.surname = Objects.requireNonNull(surname);
}

@ProtoField(number = 1)
public String getName() {
    return name;
}

@ProtoField(number = 2)
public String getSurname() {
    return surname;
}
```

Book.java

```
@ProtoFactory
public Book(String title, String description, int
publicationYear, Set<Author> authors) {
    this.title = Objects.requireNonNull(title);
    this.description = Objects.requireNonNull(description);
    this.publicationYear = publicationYear;
    this.authors = Objects.requireNonNull(authors);
}

@ProtoField(number = 1)
public String getTitle() {
    return title;
}

@ProtoField(number = 2)
public String getDescription() {
    return description;
}

@ProtoField(number = 3, defaultValue = "-1")
public int getPublicationYear() {
    return publicationYear;
}

@ProtoField(number = 4)
public Set<Author> getAuthors() {
    return authors;
}
```

If your classes have only mutable fields, then the `ProtoFactory` annotation is not required, assuming your class has a no arg constructor.

Then all that is required is a very simple `SerializationContextInitializer` interface with an annotation on it to specify configuration settings

BookContextInitializer.java

```
@AutoProtoSchemaBuilder(includeClasses = { Book.class, Author.class
}, schemaPackageName = "book_sample")
interface BookContextInitializer extends
SerializationContextInitializer {
}
```

So in this case we will automatically generate the marshaller and schemas for the included classes and place them in the schema package automatically. The package does not have to be provided, but if you utilize querying, you must know the generated package.



In Quarkus the `schemaFileName` and `schemaFilePath` attributes should NOT be set on the `AutoProtoSchemaBuilder` annotation, setting either will cause native runtime to error.

User written serialization

The previous method is suggested for any case when the user can annotate their classes. Unfortunately the user may not be able to annotate all classes they will put in the cache. In this case you must define your schema and create your own Marshaller(s) yourself.

Protobuf schema

You can supply a protobuf schema through either one of two ways.

1. Proto File

You can put the `.proto` file in the `META-INF` directory of the project. These files will automatically be picked up at initialization time.

library.proto

```
package book_sample;

message Book {
  required string title = 1;
  required string description = 2;
  required int32 publicationYear = 3; // no native Date type
  available in Protobuf

  repeated Author authors = 4;
}

message Author {
  required string name = 1;
  required string surname = 2;
}
```

2. In Code

Or you can define the proto schema directly in user code by defining a produced bean of type `org.infinispan.protostream.FileDescriptorSource`.

```
@Produces
FileDescriptorSource bookProtoDefinition() {
    return FileDescriptorSource.fromString("library.proto",
"package book_sample;\n" +
    "\n" +
    "message Book {\n" +
    "  required string title = 1;\n" +
    "  required string description = 2;\n" +
    "  required int32 publicationYear = 3; // no
native Date type available in Protobuf\n" +
    "\n" +
    "  repeated Author authors = 4;\n" +
    "}\n" +
    "\n" +
    "message Author {\n" +
    "  required string name = 1;\n" +
    "  required string surname = 2;\n" +
    "}");
}
```

User Marshaller

The last thing to do is to provide a `org.infinispan.protostream.MessageMarshaller` implementation for each user class defined in the proto schema. This class is then provided via `@Produces` in a similar fashion to the code based proto schema definition above.

Here is the Marshaller class for our Author & Book classes.



The type name must match the `<protobuf package>.<protobuf message>` exactly!

AuthorMarshaller.java

```
public class AuthorMarshaller implements
MessageMarshaller<Author> {

    @Override
    public String getTypeName() {
        return "book_sample.Author";
    }

    @Override
    public Class<? extends Author> getJavaClass() {
        return Author.class;
    }

    @Override
    public void writeTo(ProtoStreamWriter writer, Author author)
throws IOException {
        writer.writeString("name", author.getName());
        writer.writeString("surname", author.getSurname());
    }

    @Override
    public Author readFrom(ProtoStreamReader reader) throws
IOException {
        String name = reader.readString("name");
        String surname = reader.readString("surname");
        return new Author(name, surname);
    }
}
```

```
public class BookMarshaller implements MessageMarshaller<Book> {

    @Override
    public String getTypeName() {
        return "book_sample.Book";
    }

    @Override
    public Class<? extends Book> getJavaClass() {
        return Book.class;
    }

    @Override
    public void writeTo(ProtoStreamWriter writer, Book book)
    throws IOException {
        writer.writeString("title", book.getTitle());
        writer.writeString("description", book.getDescription());
        writer.writeInt("publicationYear",
            book.getPublicationYear());
        writer.writeCollection("authors", book.getAuthors(),
            Author.class);
    }

    @Override
    public Book readFrom(ProtoStreamReader reader) throws
    IOException {
        String title = reader.readString("title");
        String description = reader.readString("description");
        int publicationYear = reader.readInt("publicationYear");
        Set<Author> authors = reader.readCollection("authors", new
            HashSet<>(), Author.class);
        return new Book(title, description, publicationYear,
            authors);
    }
}
```

And you pass the marshaller by defining the following:

```

@Produces
MessageMarshaller authorMarshaller() {
    return new AuthorMarshaller();
}

@Produces
MessageMarshaller bookMarshaller() {
    return new BookMarshaller();
}

```



The above produced Marshaller method MUST return `MessageMarshaller` without types or else it will not be found.

Dependency Injection

As you saw above we support the user injecting Marshaller configuration. You can do the inverse with the Infinispan client extension providing injection for `RemoteCacheManager` and `RemoteCache` objects. There is one global `RemoteCacheManager` that takes all of the configuration parameters setup in the above sections.

It is very simple to inject these components. All you need to do is to add the `Inject` annotation to the field, constructor or method. In the below code we utilize field and constructor injection.

SomeClass.java

```

@Inject SomeClass(RemoteCacheManager remoteCacheManager) {
    this.remoteCacheManager = remoteCacheManager;
}

@Inject @Remote("myCache")
RemoteCache<String, Book> cache;

RemoteCacheManager remoteCacheManager;

```

If you notice the `RemoteCache` declaration has an additional optional annotation named `Remote`. This is a qualifier annotation allowing you to specify which named cache that will be injected. This annotation is not required and if it is not supplied, the default cache will be injected.



Other types may be supported for injection, please see other sections for more information

Querying

The Infinispan client supports both indexed and non indexed querying as long as the `ProtoStreamMarshaller` is configured above. This allows the user to query based on the

properties of the proto schema.

Query builds upon the proto definitions you can configure when setting up the `ProtoStreamMarshaller`. Either method of Serialization above will automatically register the schema with the server at startup, meaning that you will automatically gain the ability to query objects stored in the remote Infinispan Server.

You can read more about this at https://infinispan.org/docs/stable/titles/developing/developing.html#query_dsl.

You can use either the Query DSL or the Ickle Query language with the Quarkus Infinispan client extension.

Counters

Infinispan also has a notion of counters and the Quarkus Infinispan client supports them out of the box.

The Quarkus Infinispan client extension allows for Dependency Injection of the `CounterManager` directly. All you need to do is annotate your field, constructor or method and you get it with no fuss. You can then use counters as you would normally.

```
@Inject
CounterManager counterManager;
```

Near Caching

Near caching is disabled by default, but you can enable it by setting the profile config property `quarkus.infinispan-client.near-cache-max-entries` to a value greater than 0. You can also configure a regular expression so that only a subset of caches have near caching applied through the `quarkus.infinispan-client.near-cache-name-pattern` attribute.

Encryption

Encryption at this point requires additional steps to get working.

The first step is to configure the `hotrod-client.properties` file to point to your truststore and/or keystore. This is further detailed at https://infinispan.org/docs/dev/user_guide/user_guide.html#hr_encryption.

The Infinispan Client extension enables SSL by default. You can read more about this at [Using SSL With Native Executables](#).

Authentication

This chart illustrates what mechanisms have been verified to be working properly with the Quarkus

Infinispan Client extension.

Table 1. Mechanisms

Name	Verified
DIGEST-MD5	Y
PLAIN	Y
EXTERNAL	Y
GSSAPI	N
Custom	N

The guide for configuring these can be found at https://infinispan.org/docs/dev/user_guide/user_guide.html#authentication. However you need to configure these through the `hotrod-client.properties` file if using Dependency Injection.

Additional Features

The Infinispan Client has additional features that were not mentioned here. This means this feature was not tested in a Quarkus environment and they may or may not work. Please let us know if you need these added!