OSGI Compatibility

Ignite OSGi enablement involves at least two tasks:
- "bundlezation" of the Ignite Core jar, as well as the optional jars.
- OSGi-aware marshaller implementation.

Bundlezation

For the inter-package dependencies (package export/import) and other bundle metadata, OSGi relies on a number of special entries in the jar's META-INF/MANIFEST file. Those entries are usually generated semi-automatically by the Maven Bundle plugin during a Maven build. The OSGi Framework will refuse to load a jar without a valid OSGi manifest defined.

Since the Ignite's distribution includes a number of optional dependencies (such as ignite-indexing, ignite-log4j, etc) in addition to non-optional ignite-core, the proposal is to make the optional dependencies the bundle fragments hosted by the ignite-core bundle. The fact that fragments share the class loader of the host bundle should simplify inter operation between the components.

At runtime the Ignite user installs all required bundles (including ignite-core, any optional Ignite bundles as well as the application bundles) using either the standard mechanisms defined by the OSGi spec, or relying on the container's implementation-specific capabilities. For example, Apache Karaf (an OSGi implementation) offers a packaging/deloyment concept called "Feature" which roughly speaking is a list of bundles to automatically deploy when the OSGi Framework starts.

[raul.kripalani]: I've already modified the POMs to generate and package the MANIFEST.MF with the appropriate OSGi headers. The code is pushed to the ignite-1527 branch.

Feature repository for Apache Karaf

We should provide a Feature Repository to make it easier for Apache Karaf users to install Ignite and, optionally, Ignite modules. There should be one feature per module that also installs all necessary library dependencies.

Marshalling

The main problem we need to solve in order to allow Ignite OSGi enablement is the marshalling. More specifically the issue is with deserialization of the classes that are provided by the bundles other than the JDK and the Ignite bundle itself.

When the Ignite transport layer receives a message it needs to figure out how to deserialize the bytes and for that it needs to know the bundle that provides the class to be deserialized. To make things more complex, the class may contain other classes that come from other bundles, and so on recursively. In general, what is needed then is a way to map an FQN of a class to its bundle (and hence to the class loader).

And this is where ClassLoaderCodec comes into play.

On the high level the proposal is as follows:

1. During serialization, Ignite marshaller calls ClassLoaderCodec.encodeClassLoader(cls) with the class to be serialized as its only parameter. The implementation of the method may return an arbitrary object that in some way (up to the implementation) represents the class loader of the class. The encoded representation of the class loader will be serialized along with the rest of the message data. The returned object may be a primitive, a serializable POJO, or a null.
2. During deserialization, the encoded class loader (as returned by the encodeClassLoader() call during serialization) as well as the FQN of the class being deserialized are passed into ClassLoaderCodec.decodeClassLoader(fqn, encodedClassLoader) method. The implementation of the method is expected to decode and return an instance of the class loader to use for loading the class with the given FQN.

It's responsibility of the implementation to ensure that the encoded representation is sufficient to unambiguously identify the correct bundle during deserialization.

[raul.kripalani]: The naming is confusing. We are actually not transmitting classloaders. In fact, we cannot do so. What we'd like to do is transmit deserialization "hints", that are used in whatever form the marshaller deems appropriate. So if anything, I would call this class a DeserialisationHintsCodec with methods: generateHints and computeClassLoaderFromHints.

[dmitriy setrakyan]: I am not sure I see the reason for removing the word classLoader on serialization part and keeping it on deserialization. I also think that the method names should be symmetric. With that in mind, "encodeClassLoader" and "decodeClassLoader" may not be the best names, but they are consistent with each other and symmetric. My vote would be to keep the naming.

ClassLoaderCodec

The ClassLoaderCodec should be called for every Object during serialization and deserialization and should be part of the IgniteConfiguration:

```
public interface ClassLoaderCodec {
    @Nullable public Object encodeClassLoader(Class<?> cls, ClassLoader clsLdr) throws IgniteException;
    public ClassLoader decodeClassLoader(String fqn, @Nullable Object encodedClsLdr) throws IgniteException;
}
```
ClassLoaderCodec Implementations

Ignite will come with 2 OSGI class loader codecs out of the box, pessimistic and optimistic, leaving users with opportunity to provide their own custom class loader codecs as well (potentially for non-OSGI environments).

In general in OSGi, the same package may be exported by multiple bundles and therefore an FQN may not be sufficient to look up the correct class loader. In such cases, the codec implementation must employ a pessimistic approach and encode enough information (for example, the bundle symbolic name, plus the bundle version) for the deserializer to be able to resolve the FQN to the correct class loader. Such implementation will work for all use cases, but it introduces some overhead and increases the size of the serialized messages.

However, for the applications that can enforce one-to-one mapping of packages to bundles, a simplified (optimistic) approach can be used instead. With this approach, no encoding of the class loader is required (encodeClassLoader() returns null), and only the FQN is used for decoding of the class loader.

[raul.kripalani]: I don't like transmitting bundle symbolic names over the wire, as it couples the serialising party with the deserialising party, forcing both to contain the class inside the same bundle. As I said in the mailing list, making this assumption would be a short-sighted strategy, as users may be sharing caches across applications across multiple containers, where classes live in different bundles in different containers.

I also don't think it's necessary. We just need the package name + package version. An OSGi container cannot expose the same package under the same version number twice, so the tuple (package name, package version) is enough to unambiguously locate the Bundle that exports our class.

Now, what we need to do is determine HOW we locate the Bundle. I have two ideas in mind:

1. Create a custom OSGi Manifest header Ignite-Export-Package that lists the packages to be made available to Ignite for deserialisation purposes. Our Activator would register a BundleTracker that introspects Bundle installations and maintains a Map between (package name, package version) => Bundle, of only those bundles where the user has expressly indicated that there are packages to be made available to Ignite.
2. Avoid the header and use a logic like the bundle:find-class command in Karaf: https://github.com/apache/karaf/blob/master/bundle/core/src/main/java/org/apache/karaf/bundle/command/FindClass.java. This logic queries all bundles in the container to locate the package/class. We would build a memory cache (ConcurrentHashMap) to avoid performing this lookup more than once for the same package. We also need a BundleTracker to clear the package cache for bundles that are uninstalled.

With either of these approaches, I think we don't need pessimistic and/or optimistic strategies. Just a single strategy would be enough.

Here’s how the pessimistic codec implementation might look like (in pseudo-code):

```java
public class ClassLoaderPessimisticCodec implements ClassLoaderCodec {
    public ClassLoaderPessimisticCodec() {} {

        @Nullable public Object encodeClassLoader(Class<?> cls, ClassLoader clsLdr) throws IgniteException {
            // TODO
            return bundleName + bundleVersion;
        }

        public ClassLoader decodeClassLoader(String fqn, @Nullable Object encodedClsLdr) throws IgniteException {
            // TODO: get class loader for a bundle based on bundleName and version.
            ...
        }
    }
}
```

Here's how the optimistic (opportunistic :)))) codec implementation might look like:
public class ClassLoaderOptimisticCodec implements ClassLoaderCodec {

    public ClassLoaderOptimisticCodec() {}

    @Nullable public Object encodeClassLoader(Class<?> cls, ClassLoader clsLdr) throws IgniteException {
        return null;
    }

    public ClassLoader decodeClassLoader(String fqn, @Nullable Object encodedClsLdr) throws IgniteException {
        // TODO:
        // Iterate through all the bundles and pick the first one
        // that can load the class. Once found, cache the class loader
        // for faster lookups going forward.
        ...
    }
}

Implementation strategies

Assumptions

First of all the both approaches imply that your cluster is consistent and contains the same version of the bundles on all the nodes. This can be see a a valid assumption in order to ensure the consistency of your computation tasks. If you want to be able to work it in a more non deterministic approach then we have to introduce yet another strategy. But first let focus assume that the bundles are equals on the entire cluster.

Supported version of OSCi

TBD >= 5.0

Pessimistic Codec

On the write side this approach require you to capture the bundle symbolic name and its version. This is something easy to do as in OSGi all classloader except the system classloader implements the BundleReference. The pessimistic codec can look like that:

```java
public class ClassLoaderPessimisticCodec implements ClassLoaderCodec {

    private static final byte FRAMEWORK_CLASS_LOADER_ID = 0;
    private static final byte IGNITE_CLASS_LOADER_ID = 1;
    private static final byte BOOT_CLASS_LOADER_ID = 2;
    private static final byte BUNDLE_CLASS_LOADER_ID = 4;

    private static final ClassLoader FRAMEWORK_CLASS_LOADER = Bundle.class.getClassLoader();

    private final PackageAdmin packageAdmin;

    public ClassLoaderPessimisticCodec(PackageAdmin packageAdmin) {
        this.packageAdmin = packageAdmin;
    }

    @Nullable @Override
    public Object encodeClassLoader(Class<?> cls) throws IgniteException {
        ClassLoader classLoader = cls.getClassLoader();
        if (isIgniteClass(classLoader)) {
            return ClassLoaderDesc.newIgniteClassLoaderDesc();
        }
        if (isFrameworkClassLoader(cls.getClassLoader())) {
            return ClassLoaderDesc.newFrameworkClassLoader();
        }
        Bundle bundle = FrameworkUtil.getBundle(cls);
        if (bundle != null) {
            return ClassLoaderDesc.newBundleClassLoaderDesc(bundle);
        }
    }
}
```
return ClassLoaderDesc.newBootClassLoader();
}

@Override
public ClassLoader decodeClassLoader(String fqn, ClassLoader clsLdr, @Nullable Object encodedClsLdr) throws IgniteException {
    ClassLoaderDesc classLoaderDesc = (ClassLoaderDesc) encodedClsLdr;
    switch (classLoaderDesc.classLoaderId) {
        case BOOT_CLASS_LOADER_ID:
            return clsLdr;
        case FRAMEWORK_CLASS_LOADER_ID:
            return FRAMEWOR_CLASS_LOADER;
        case IGNITE_CLASS_LOADER_ID:
            return ClassLoaderCodec.class.getClassLoader();
        case BUNDLE_CLASS_LOADER_ID:
            Bundle[] bundles = packageAdmin.getBundles(classLoaderDesc.bsn, classLoaderDesc.version);
            if (bundles == null) {
                throw new IgniteException("No bundle found: " + classLoaderDesc.bsn + ":" + classLoaderDesc.version);
            }
            try {
                //highest ranking bundle
                return bundles[0].loadClass(fqn).getClassLoader();
            } catch (ClassNotFoundException e) {
                throw new IgniteException(e);
            }
        default:
            throw new IgniteException("Unsupported class loader description type: " + classLoaderDesc.classLoaderId);
    }
}

static final class ClassLoaderDesc implements Externalizable {
    private String version;
    private String bsn;
    private byte classLoaderId;
    public ClassLoaderDesc() {}
    public ClassLoaderDesc(byte classLoaderId) {
        this.classLoaderId = classLoaderId;
    }
    public ClassLoaderDesc(Bundle bundle) {
        this.classLoaderId = BUNDLE_CLASS_LOADER_ID;
        this.bsn = bundle.getSymbolicName();
        this.version = bundle.getVersion().toString();
    }

    @Override
    public void writeExternal(ObjectOutput out) throws IOException {
        out.write(classLoaderId);
        if (classLoaderId == BUNDLE_CLASS_LOADER_ID) {
            out.writeUTF(bsn);
            //can be optimized
            out.writeUTF(version);
        }
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        classLoaderId = in.readByte();
        if (classLoaderId == BUNDLE_CLASS_LOADER_ID) {
            this.bsn = in.readUTF();
            this.version = in.readUTF();
        }
    }

    @Override
    public void readExternal(ObjectInput in) throws IOException, ClassNotFoundException {
        classLoaderId = in.readByte();
        if (classLoaderId == BUNDLE_CLASS_LOADER_ID) {
            this.bsn = in.readUTF();
            this.version = in.readUTF();
        }
    }
}
static ClassLoaderDesc newIgniteClassLoaderDesc() {
    return new ClassLoaderDesc(IGNITE_CLASS_LOADER_ID);
}

public static ClassLoaderDesc newBundleClassLoaderDesc(Bundle bundle) {
    return new ClassLoaderDesc(bundle);
}

public static ClassLoaderDesc newFrameworkClassLoader() {
    return new ClassLoaderDesc(FRAMEWORK_CLASS_LOADER_ID);
}

public static ClassLoaderDesc newBootClassLoader() {
    return new ClassLoaderDesc(BOOT_CLASS_LOADER_ID);
}

Disclaimer: this implementation is not functional and not optimized is purpose it to show how it can be done. Here we are using the PackageAdmin service which is deprecated but really simple to demonstrate the purpose.

⚠️ It seems to be more interesting to get the ClassLoader clsLdr as a parameter of the decode method instead of the encode one.

**Optimistic codec**

In this strategy we start with a more strict assumption: packages of all serialized classes come from one and only one bundle.

You may think this option is more simple but it is not true.

TODO