gridgo Documentation

dungba88

Oct 30, 2019

General

Welcome to the official documentation of Gridgo, a platform to create distributed system easier. We provides the manual, tutorials and examples on this documentation.

Note: All contributions are welcome. Make sure you read the contribution guideline before start making changes.

The table of contents below and in the sidebar should let you easily access the documentation for your topic of interest. You can also use the search function in the top left corner.

Tip: If you are new to Gridgo, please read the first section about basic principles and terminologies first.

The main documentation for the site is organized into the following sections:

CHAPTER 1

Introduction

This page aims at providing basic principles and terminologies of Gridgo. It is recommended if you are new to Gridgo.

1.1 About Gridgo

Gridgo is a platform to create distributed systems easier with asynchronous I/O connectors and event driven programming. Gridgo handles the heavylift work of I/O and thread routing so you can focus on designing network topologies and implementing business logic.

1.2 Gridgo Principles

Everyone has principles, so does Gridgo. Gridgo follows and strongly recommends several principals, which are:

Fluent API Most Gridgo APIs are fluent. You can perform successive operations within a single method calls chain.

Asynchronous over Synchronous It is as straightforward as it sounds. Most of the operations in Gridgo are asynchronous, even for remote-procedure calls. Asynchronous operations will give you a `Promise`, which you can be notified when it is fulfilled or rejected. Some operations which need response from you will give you a `Deferred` that you can fulfill or reject it. Your method also should not block.

Think topology over protocol Instead of focusing on the actual protocol, you should focus on designing the network or computing topology. All I/O operations are abstracted using `Connector` or `Gateway`.

1.3 Gridgo Components

The main components of Gridgo are:

Connector This is the most basic abstraction level in Gridgo. It provides easy-to-use I/O connection (sending and receiving messages). Each type of connector is uniquely represented by an `endpoint`. An example might be *"kafka:mytopic?brokers=127.0.0.1"*. Connector consists of `Producer` and `Consumer`

Producer The outgoing part of `connector`. It allows you to send messages to the remote service with *fire-and-forget* or *RPC* styles.

Consumer The incoming part of `connector`. It allows you to subscribe and handling messages. Consumer can be two-way, which means it can also send response back to the caller.

Gateway This is the bridge between `Connector` and application logic. It allows you to subscribe for messages from and send messages to the underlying connectors. Gateways are always accessed by name.

Application Context This is the highest level component, which will connect all other components, like opening gateways, resolving connectors, etc. You can create any number of contexts inside a single JVM process.

CHAPTER 2

Installing Gridgo

Gridgo requires Java 11 so you should make sure JDK 11 is installed. Gridgo is tested with `Oracle JDK` but any other JDK should be fine.

Gridgo itself can be easily installed using Maven. It is also split into several independent components that can be installed separately:

Install gridgo-core

This is the recommended package, which provides the most functionalities of Gridgo (`Gateway`, `Application Context`)

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-core</artifactId>
    <version>0.1.2</version>
</dependency>
```

Install gridgo-connector-core

Install this if you want to only work with the I/O abstraction layer (`Connector`)

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-connector-core</artifactId>
    <version>0.1.0</version>
</dependency>
```

Note: To work with a specific connector type (e.g *gridgo-kafka*) you also need to install it separately:

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-kafka</artifactId>
    <version>same as gridgo-connector-core</version>
</dependency>
```

A full list of supported connectors can be found on the GitHub repository

CHAPTER 3

Creating first application

This article will help you creating your first Gridgo application. This simple application will do the following:

- Create a new application context
- Open a gateway and attach a HTTP server to it (using *gridgo-vertx-http*)
- Start listening for incoming HTTP requests and return the same as responses.

Tip: The full source code for this example can be found in the *Examples* section

The entry-point of a Gridgo application is the `GridgoContext`. A GridgoContext will act as a standalone component which will have its own configuration and be started/stopped independently regardless of where it's running. While a JVM process is a physical entity, a GridgoContext is a logical one, and in fact, you can have multiple instances of GridgoContext inside a single JVM process.

GridgoContext can be created using a `GridgoContextBuilder`, which currently supports `DefaultGridgoContextBuilder`

```
// create the context using default configuration
var context = new DefaultGridgoContextBuilder().setName("application").build();
```

GridgoContext allows you to open `Gateway`. Gateway is the abstraction level between the I/O layer (`Connector`) and business logic code. It allows you to write code in event-driven paradigm. You can also think of Gateway as the bridge between your application business logic and the external (remote or local) endpoints.

Gateways are asynchronous in nature, which all interactions are handled using Promise.

The following code will open a new gateway, attach an I/O connector to it and subscribe for incoming messages.

```
var gateway = context.openGateway("myGateway")
                     .attachConnector("vertx:http://127.0.0.1:8080/") // attach a web␣
↪server connector
                     .subscribe(this::handleMessages) // subscribe for incoming␣
↪messages
                     .finishSubscribing().get();
```

More about Connector and available endpoints can be found here

handleMessages is actually a implementation of `Processor`, which will take 2 arguments: a `RoutingContext` containing information about the current request and the GridgoContext the request is associated with.

```
private void handleMessages(RoutingContext rc, GridgoContext gc) {
    var msg = rc.getMessage();
    var deferred = rc.getDeferred();

    // using the same request body as response
    deferred.resolve(Message.newDefault(Payload.newDefault(msg.getPayload.
↪getBody())));
}
```

After you have configured the context, you need to call its `start()` method, which will in turn starting gateways. If the attached connector supports a `Consumer`, it will listen for incoming messages and route them to the matching Processors.

```
context.start();

// Register a shutdown hook to stop the context
Runtime.getRuntime().addShutdownHook(new Thread(context::stop));
```

After you have run the application (e.g using a *public static void main(String[] args)* method), you can access the application in http://localhost:8080

Well done! You have created your first Gridgo application with just a few lines of code. More hand-on tutorials can be found at the /tutorials/index section, or you can go to the next section for more advanced topics.

CHAPTER 4

Bean Basics

4.1 Overview

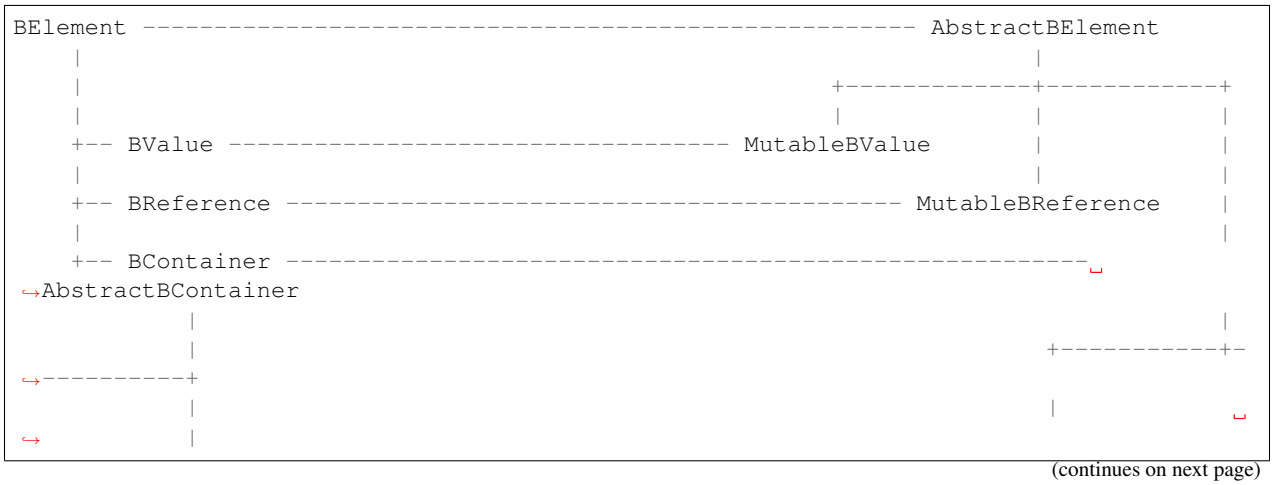
Beans are the abstract data structure of Gridgo aim to transparent data format and make easier to sending via network.

Bean's structure is json-like, which support reference in additional.

To use bean in your project, just add maven dependency:

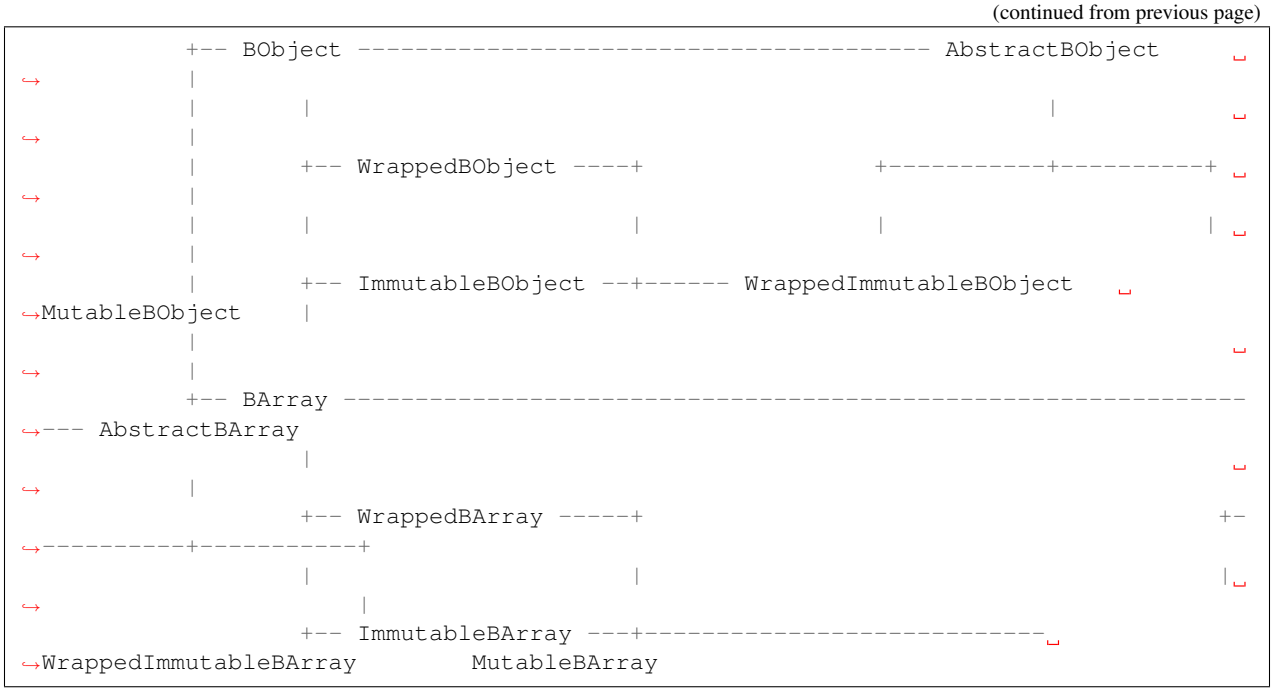
```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-bean</artifactId>
    <version>${gridgo.bean.version}</version>
</dependency>
```

4.2 Hierarchy



(continues on next page)

(continued from previous page)



4.3 Definition

BElement is the ancestor of all bean types. It's useful if you want to create a bean from any type.

BReference wraps any object as a reference.

BValue wraps any value which is primitive: *Boolean*, *Character*, *String*, *Number* (byte, short, int, long, float, double, BigInteger, BigDecimal), *raw binary* (byte[]). BValue can also convert any primitive to other (primitive) types.

BContainer base class for any BElement type which can contains other BElement.

BObject defines a key-value data structure.

BArray defines a sequence data structure.

4.4 Usage

Using static methods define in top interface to create correlative instance:

4.4.1 Mutable instances

- `BReference.of(data)`: create a mutable instance of BReference
- `BValue.of(data)`: create a mutable instance of BValue which wrap data as its source.
- `BObject.ofEmpty()`, `BObject.of(Map<?,?>)`: create a mutable instance of BObject which is empty or auto convert map to a BObject
- `BArray.ofEmpty()`, `BArray.of(collection or array)`: create a mutable instance of BArray which is empty or auto convert (collection | array) into a BArray

in case you don't know which type of data want to convert to BElement, use: `BElement.ofAny(data)`

4.4.2 Immutable instances

- `BObject.wrap(map)`: wrap the input map in a `WrappedImmutableBObject`
- `BArray.wrap(collecion or array)`: wrap the input (collection or array) in a `WrappedImmutableBArray`

In case you don't know which kind of data which want to wrap, use: `BElement.wrapAny(data)`

4.4.3 Working with pojo

If you have a pojo object and want to convert it to BObject, you can use `BObject.ofPojo()`.

4.5 Serialization

BElement support multi kind of data serialization:

4.5.1 Built-in

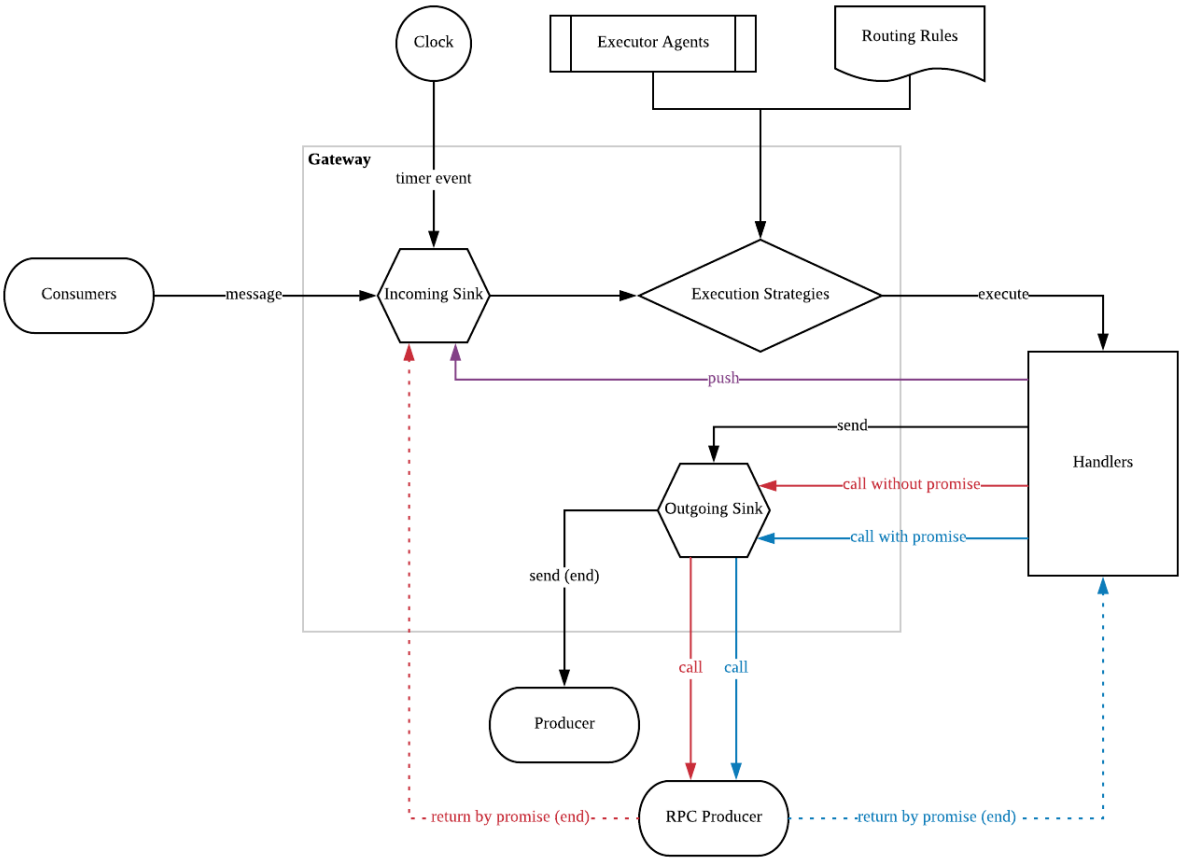
- **JSON** can be accessed by `BElement.toJson()` and `BElement.fromJson(...)`
- **XML** can be accessed by `BElement.toXml()` and `BElement.fromXml(...)`
- **Msgpack** can be accessed by `BElement.toBytes()` and `BElement.fromBytes(...)` (*msgpack* set as default binary serializer if the system property name *gridgo.bean.serializer.binary.default* is unset)

CHAPTER 5

Gateway Basics

Gateway is a level of abstraction above Connector. It is the bridge between Connector and Handler, making it easier to create event-driven application. Several connectors can be attached to one gateway, after which any requests routed to the gateway will be multicasted to the attached connectors' producers.

The complete architecture of gateways is:



Gateway comprises of Incoming Sink and Outgoing Sink:

5.1 Incoming sink

This sink accepts events from various types of dispatcher, including *consumers*, *timers*, *producer* and *programmatically from handlers*.

Consumers: Consumers read messages from a datasources (file, message queues, etc.) or accept messages from external system (e.g using a web server), then convert it to a *Message* and put into the Gateway's incoming sink. Consumer can sometimes be a timer which schedules messages to be generated and be put into the incoming sink. Each timer will accept a *Supplier<Message>*.

Producers: Response from the producers can be put it into the incoming sink, making them available to be consumed by other handlers.

Handlers: Handlers can manually create the message and programmatically put it into the incoming sink.

After a message is put into the incoming sink, it will be routed to different handlers based on the configured policies and rules. The corresponding handlers will then be executed with the following parameters:

***executionContext*:** contains the *Message* to be consumed, the *DeferredObject* to be fulfilled and the caller (the gateway that this message comes from). Only the first call to resolve() or reject() will be considered, after that all subsequent calls are ignored.

applicationContext contains all application-related information, including access to gateways, configurations, etc.

5.2 Outgoing sink

5.3 Execution strategies

5.4 Using gateways

To create a gateway, attach connectors and bind handlers

```
var appContext = new GridgoContext();
var connector = appContext.openGateway("Orders")
                          .attachConnector("kafka:orders?brokers=127.0.0.1")
                          .attachConnector(someConnector)
                          .subscribe(handler1).withPolicy(somePolicy).
↪finishSubscribing()
                          .subscribe(handler2).when(someCondition).with(someStrategy).
↪finishSubscribing();
```

Attempting to open gateway with the same name will cause a *DuplicateGatewayException*

To access an opened gateway:

```
var gateway = appContext.findGateway("Orders"); // return a Optional<Gateway>
if (gateway.isPresent()) {
    // do some works with the gateway
}
```

or access the list of opened gateway

```
var gateways = appContext.getGateways(); // return a List<Gateway>
```

CHAPTER 6

Bean Advanced

This section will discuss about advanced topics when using Gridgo Bean

6.1 Thread-safe

By default, all BElement created using default static methods `of[...]()` are always *non* thread-safe.

To create a thread-safe BElement, use `BObject.withHolder(<holder_map>)` or `BArray.withHolder(<holder_list>)`. Where the parameter is instance of a thread-safe type (e.g *ConcurrentHashMap*).

6.2 Pluggable serialization

Every serialization format implements *BSerializer*, include all of the built-in ones.

For example, instead of using `BElement.ofJson(json)` you can also use a more generalized approach: `BElement.fromBytes(bytes, "json")`. The second parameter is the name of the *BSerializer* you registered in the Serialization Registry (more on this later). `json`, `xml`, `default` or `msgpack` is built-in, which means they are implicity registered.

To write you own serializer, create a class which implements *BSerializer* and annotate it by `@BSerializationPlugin(name)`. After that, register it with *BSerializerRegistry* by calling:

```
BFactory.DEFAULT.getSerializerRegistry().scan(<package_name_of_your_serializer>);
```

or

```
BFactory.DEFAULT.getSerializerRegistry.register(<name>, <BSerializer instance>);
```

By default `BFactory.DEFAULT.getSerializerRegistry()` auto scan package `io.gridgo.serialization`, so that if your custom serializer located in that package and loaded in the same class loader with BElement, you don't need to call register.

Note: your custom *BSerializer* must be thread-safe.

6.3 Default binary serializer

Bean using a system property named `gridgo.bean.serializer.binary.default` to take default binary serializer name, if it's unset, `msgpack` will be used.

Default serializer will be used in `toBytes()`, `writeBytes(...)` and `fromBytes(...)` methods.

You can, for example, change the default binary format to avro by providing the correct system property value when starting the JVM:

```
-Dgridgo.bean.serializer.binary.default=avro
```

6.4 Schema and schema-less

6.4.1 Schema

There are a lot of binary serialization format which based on a *schema* - generally understood as a value object.

To serialize an object with a schema serialization format, the object's type must be registered with the serializer.

Generally, all schema format supports 2 modes: single and multi. You use single format if your whole application only works with one schema (which is usually not the case). If you works with multiple schemas, there are 2 ways you can do:

- Use multi schemas mode: Using this you need to register the schemas with the serializer. The serialized data will then be prepended with a *type* header (commonly using 4 bytes integer), which might not be inter-operable.
- Use many single-schema mode: Using this you need to create individual serializer for each schema. The advantage is that the data won't be prepended with a *type* header, thus it's more inter-operable.

There are 2 interfaces for *schema serialization*: `io.gridgo.bean.serialization.MultiSchemaSerializer<S>` and `io.gridgo.bean.serialization.SingleSchemaSerializer<S>`

6.4.2 Schema-less

JSON, *XML*, *msgpack* (all of the built-in formats) are schema-less, which mean they don't need a pre-defined schema class.

6.5 Pre-support serialization

6.5.1 Gson

By default, *json* serializer using `json-smart` lib, but if you have your own reason to use `gson`, you can do it by add maven dependency:

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-bean-gson</artifactId>
    <version>x.x.x</version>
</dependency>
```

Then, you can serialize using `gson` by calling: `BElement.toBytes('gson');`

6.5.2 Protobuf

Protobuf is a popularly serialization format. Gridgo-bean already support it with a dependency:

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-bean-protobuf</artifactId>
    <version>x.x.x</version>
</dependency>
```

Protobuf serializer support 2 modes:

- Single schema:

First you need to register the schema once when you start the application:

```
ProtobufSingleSchemaSerializer protobufSerializer = BFactory.DEFAULT.
↪getSerializerRegistry().lookup(ProtobufSingleSchemaSerializer.NAME);
protobufSerializer.setSchema(Person.class);
```

Then you can start using it:

```
// create a person instance
Person p = createPerson();
BElement ele = BElement.ofAny(p);

// serialize the person instance using Protobuf format
byte[] bytes = ele.toBytes(ProtobufSingleSchemaSerializer.NAME);

// deserialize it
BElement unpackedEle = BElement.ofBytes(bytes, ProtobufSingleSchemaSerializer.NAME);
Person p2 = unpackedEle.asReference().getReference();

// the two should be equals
Assert.assertEquals(p, p2);
```

- Multi schema:

First you need to register the schema once when you start the application. You need to associate the schema with a *type* (an integer):

```
ProtobufMultiSchemaSerializer protobufSerializer = BFactory.DEFAULT.
↪getSerializerRegistry().lookup(ProtobufMultiSchemaSerializer.NAME);
protobufSerializer.registerSchema(Person.class, 1);
```

Then you can start using it as normal:

```
Person p = createPerson();

BElement ele = BElement.ofAny(p);
byte[] bytes = ele.toBytes(ProtobufMultiSchemaSerializer.NAME);

BElement unpackedEle = BElement.ofBytes(bytes, ProtobufMultiSchemaSerializer.NAME);
Person p2 = unpackedEle.asReference().getReference();

assertEquals(p, p2);
```

`Person` is a protobuf generated class.

Note: you must register the schema class before using *protobuf* serialization format. Only *BReference* contains registered *schema* can be serialized/deserialized

---

6.5.3 Avro

Like protobuf, Avro is also a widely-used serialization format. To use it, add below lines to your pom.xml:

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-bean-avro</artifactId>
    <version>x.x.x</version>
</dependency>
```

Avro serializier also support 2 modes:

- Single schema:

```
AvroSingleSchemaSerializer avroSerializer = BFactory.DEFAULT.getSerializerRegistry().
↪lookup(AvroSingleSchemaSerializer.NAME);
avroSerializer.setSchema(Person.class);

Person p = createPerson();
byte[] bytes = BElement.ofAny(p).toBytes(AvroSingleSchemaSerializer.NAME);

BElement unpackedEle = BElement.ofBytes(bytes, AvroSingleSchemaSerializer.NAME);
Person p2 = unpackedEle.asReference().getReference();

assertEquals(p, p2);
```

- Multi schema:

```
AvroMultiSchemaSerializer avroSerializer = BFactory.DEFAULT.getSerializerRegistry().
↪lookup(AvroMultiSchemaSerializer.NAME);
avroSerializer.registerSchema(Person.class, 1);

Person p = createPerson();
byte[] bytes = BElement.ofAny(p).toBytes(AvroMultiSchemaSerializer.NAME);

BElement unpackedEle = BElement.ofBytes(bytes, AvroMultiSchemaSerializer.NAME);
Person p2 = unpackedEle.asReference().getReference();

assertEquals(p, p2);
```

where `Person` is a avro generated class.

Note: you must register the schema class before use *avro* serialization format. Only *BReference* contains registered *schema* can be serialized/deserialized

6.6 Write out binary

To work with I/O, data should be written to an output stream. There are 2 ways to do that:

1. convert to byte[] using `BElement.toBytes()` then append that bytes to output stream.
2. write directly to output stream using `BElement.writeBytes(outputStream)`.

The second way is highly recommended because it save one mem-copying and will make your code faster.

CHAPTER 7

Gateway Advanced

This section will discuss about advanced topics when using Gateway

7.1 Multiple connectors per gateway

It is possible to attach multiple connectors to a single Gateway. Doing so will make incoming messages from all attached Connectors to be routed to the Processors. One more interesting thing is that when you send messages to Gateway (using either *send()*, *sendWithAck()*, *call()* or *callAndPush()*), the messages will also be multiplexed to all Connectors.

So what is the response if you make RPC calls to a Gateway having multiple Connectors? Well, Gridgo allows you choose the strategy to compose the response, using *ProducerTemplate*. There are 3 built-in types of *ProducerTemplate*:

- **SingleProducerTemplate**: which will keep the first Connector response and discard all others, this is the default template
- **JoinProducerTemplate**: which will merge all responses into a single *MultipartMessage*
- **MatchingProducerTemplate**: similar to *JoinProducerTemplate*, but allows you to use a *Predicate* to filter what Connector to be called. Responses are also merged into a single *MultipartMessage*

7.2 Returning responses to connectors

Many Connectors require responses or acknowledgements from Processors, e.g in a HTTP server, you need to send the response back to client, or in Kafka you need to send acknowledgement back to KafkaConnector, so it will commit the message. This is done using the *Deferred* object in *RoutingContext*

```
private void handleMessages(RoutingContext rc, GridgoContext gc) {
    try {
        // do some work to get the response
        // Gridgo favors asynchronous over synchronous, so your method shouldn't block
        rc.getDeferred().resolve(response);
```

(continues on next page)

(continued from previous page)

```
    } catch (Exception exception) {
        // or reject the request with some exception
        rc.getDeferred().reject(exception);
    }
}
```

Note: Only the first call to either *resolve()* or *reject()* will work. Subsequent calls will be ignored.

7.3 Configuring an execution policy

When subscribing to Gateway's incoming messages, you can optionally configure an **Execution Policy**. Execution policies allow you to control in which condition and how the processors will be executed. For example, to only execute a processor in a particular condition:

```
var gateway = context.openGateway("myGateway")
                     .attachConnector("kafka:mytopic")
                     .subscribe(this::handleMessages)
                     .when("payload.body.data > 1") // only execute the Processor if␣
↪payload body is numeric and greater than 1
                     .finishSubscribing().get();
```

Or, to execute the processor using a particular strategy, *ExecutorService* for instance:

```
var gateway = context.openGateway("myGateway")
                     .attachConnector("kafka:mytopic")
                     .subscribe(this::handleMessages)
                     .when("payload.body.data > 1")
                     .using(new ExecutorExecutionStrategy(8))
                     .finishSubscribing().get();
```

Calling *.subscribe()* will actually return a `HandlerSubscription`, which you can call *.when()* and *.using()* upon. This is called fluent-style API. To return the flow back to Gateway, you will call *.finishSubscribing()*.

7.4 Instrumenters

Instrumentering is a mechanism to alter a class behavior without directly modifying it. There are two types of instrumenters in Gridgo: Execution Strategy Instrumenter and Producer Instrumenter.

7.4.1 Execution Strategy Instrumenters

Execution strategy instrumenters are used with gateway processors to alter their behaviors. When you attach an instrumenter to a processor, all handling will be routed to the it instead of the processor. Some examples of instrumenters might be:

- Calculate the throughput and latency of a processor via Prometheus integration.
- Apply access control going through a processor.

There are several instrumenters already available for use, e.g:

- WrappedExecutionStrategyInstrumenter: Wrap several instrumenters into a single one. Wrapped instrumenters will be executed sequentially.
- Prometheus-integration instrumenters: Integrate with Prometheus functionality. Can be used to calculate throughput and latency of a processor.

7.4.2 Producer Instrumenters

Producer instrumenters are used with producers to alter their behaviors. Some examples of producers instrumenters might be:

- Calculate the throughput and latency of a producer (e.g a JDBC producer)
- Add a cache layer to the producer
- Add a mockup layer to the producer

7.5 Transaction in Processors

Gridgo supports asynchronous transaction through messaging. If a gateway is attached with a connector which supports Transaction (more on this later), a Processor can use it to call requests, commit or rollback the transaction.

Transactions are done through *io.gridgo.connector.support.transaction.Transaction* object. To make gateway agnostic of transaction, the creation of the Transaction object is done via Message (see Behind the scene section). A Processor can also implements *TransactionProcessor* interface, which provides utility methods for handling transactions.

7.5.1 Creating and processing transactions

There are several ways to create and process transactions. All of the following code are valid and are exactly the same. *createTransaction* and *withTransaction* methods are provided by *TransactionProcessor*

1. Create, commit and rollback transaction manually

```
createTransaction(gateway).done(transaction -> {
    // use the Transaction object to query and commit/rollback manually
    transaction.callAny("insert into some_table values(..)")
                .pipeDone(result -> doSomethingWithResult())
                .done(result -> transaction.commit())
                .fail(ex -> transaction.rollback());
}).fail(ex -> logger.error("Cannot create transaction", ex));
```

2. Create transaction automatically but commit/rollback manually

```
// create transaction automatically but still commit/rollback manually
withTransaction(gateway, transaction -> {
    transaction.callAny("insert into some_table values(..)")
                .pipeDone(result -> doSomethingWithResult())
                .done(result -> transaction.commit())
                .fail(ex -> transaction.rollback())
}).fail(ex -> logger.error("Cannot create transaction", ex));
```

3. Create transaction manually and use provided deferred to commit/rollback

```
// create transaction automatically and use provided deferred to commit/rollback
// the transaction will be committed when deferred is resolved, and rolled back
// when it is rejected
withTransaction(gateway, (transaction, deferred) -> {
    transaction.callAny("insert into some_table values(..)")
               .pipeDone(result -> doSomethingWithResult())
               .forward(deferred);
}).fail(ex -> logger.error("Cannot create transaction", ex));
```

4. Create transaction automatically and return a promise to commit/rollback

```
// create transaction automatically and return a promise to let Gridgo knows
// when to commit/rollback the transaction.
withTransaction(gateway, transaction -> {
    return transaction.callAny("insert into some_table values(..)")
                      .pipeDone(result -> doSomethingWithResult());
}).fail(ex -> logger.error("Cannot create transaction", ex));
```

CHAPTER 8

Context Advanced

This section will discuss about advanced topics when using GridgoContext

8.1 Using configuration with Gridgo

Gridgo supports two types of configuration: a *key-value* config and an *object-based* config. Key-value configurations are usually good for storing application settings, similar to a properties file, or bean registration, similar to Spring bean. Object-based configurations on the other hand suitable for storing complex data structure, like when configuring *GridgoContext*. Refer to each section below to get more details about each type of configuration.

8.1.1 Key-value configuration

Key-value configurations in Gridgo are called *Registry*. A registry stores a mapping between a String and an arbitrary object. Two most basic operations are *lookup()* and *register()*, e.g:

```
var registry = new SimpleRegistry();
registry.register("numbers", new int[] {1, 2, 3});
var numbers = registry.lookup("numbers");
```

You can also pass a type to the *lookup* method, so result will be automatically casted the specified type:

```
var numbers = registry.lookup("numbers", int[].class);
```

And substitute placeholders in strings using values from Registry:

```
var someString = "mongodb://${mongodb.host}:${mongodb.port}";
var parsedString = registry.substitute(someString);
```

Some of the supported registries are:

- *SimpleRegistry*: a registry backed by a HashMap
- *PropertiesFileRegistry*: a registry backed by a properties file

- *SystemEnvRegistry*: a registry which corresponds to the system environment variables
- *SystemPropertyRegistry*: a registry which corresponds to the Java properties (e.g when using *-D* option)
- *MultiSourceRegistry*: a registry which contains other registries, so you can use multiple registries as if it was a single one.

There are some registries supported in *gridgo-extras*:

- *SpringRegistry*: a registry which is backed by Spring *ApplicationContext*
- *TypeSafeRegistry*: a registry which is backed by typesafe/config

8.1.2 Object-based configuration

Object-based configurations in Gridgo are called *Configurator*. They are a bit more advanced than *Registry* which can support hot-reload. But they don't support looking up individual key inside the configurations. Rather they will return the whole configuration as a *BElement*.

```
// create an instance of configurator using TypeSafe
var configurator = TypeSafeConfigurator.ofResource("application.conf");

// register for configuration event
configurator.subscribe(event -> {
  if (event.isLoaded())
    System.out.println("Event loaded: " + event.asLoaded().getConfigObject());
  else if (event.isReloaded())
    System.out.println("Event reloaded: " + event.asLoaded().getConfigObject());
  else if (event.isFailed())
    event.asFailed().getCause().printStackTrace();
});

// start the configurator
configurator.start();
```

The configurator needs more work than Registry because it can support hot-reload and remote configuration (e.g using a Database or Zookeeper)

Some currently supported configurator

- *TypeSafeConfigurator*: configurator backed by TypeSafe, supporting JSON, HOCON
- *YamlConfigurator*: configurator using YAML format
- *JsonConfigurator*: configurator using JSON format

Tip: Most configurators are extended from *ReplayEventDispatcher*, that means you can subscribe whenever you want, all events will be replayed every time you subscribe.

8.2 Creating GridgoContext from configuration

It is very convenient to create GridgoContext using configuration, whether it is HOCON, JSON or YAML. The code is very simple:

```
// create a configurator using TypeSafe, which supports JSON and HOCON
var configurator = TypeSafeConfigurator.ofResource("gridgo-context.conf");

// create the context using the configurator
var context = new ConfiguratorContextBuilder().setRegistry(registry)
                                              .setConfigurator(configurator)
                                              .build();
```

Following are an example of a conf file:

```
# the application name
applicationName = "helloworld"

# list of gateways
gateways {

    # define a gateway with name "test" and one subscriber
    test.subscribers += "class:io.gridgo.example.TestProcessor"

    # add another gateway with name "another" and one subscriber
    # with a custom execution strategy and condition
    another.subscribers += {
        processor = "class:io.gridgo.example.AnotherProcessor"
        executionStrategy = "bean:myExecutionStrategy"
        condition = "payload.body.data not empty"
    }

    # add another gateway with 2 connectors and autoStart false
    alsoAnother {
        autoStart = false
        connectors += "kafka:topic1?brokers=localhost:9092"

        # this connector will have a custom ConnectorContextBuilder
        connectors += {
            endpoint = "vertx:http://localhost:8080"
            contextBuilder = "bean:myConnectorContextBuilder"
        }
        subscribers += "class:io.gridgo.example.AlsoAnotherProcessor"
    }
}

# list of components
components += "bean:myComponent"
```

This is the same configuration using YAML

```
applicationName: "helloworld"

gateways:
    test:
        subscribers:
            - "class:io.gridgo.example.TestProcessor"
    another:
        subscribers:
            - processor: "class:io.gridgo.example.AnotherProcessor"
              executionStrategy: "bean:myExecutionStrategy"
              condition: "payload.body.data not empty"
```

(continues on next page)

(continued from previous page)

```
    alsoAnother:
        autoStart: false
        connectors:
            - "kafka:topic1?brokers=localhost:9092"
            - endpoint: "vertx:http://localhost:8080"
              contextBuilder: "bean:myConnectorContextBuilder"
        subscribers:
            - "class:io.gridgo.example.AlsoAnotherProcessor"
components:
    - "bean:myComponent"
```

As you can see the syntax very flexible: for *subscribers*, *connectors* you can use either String or Object. The syntax is based on HOCON format, a superset of JSON, which is optimized for human.

For *processor*, you can either use a bean or a class. If you use a bean, it must be registered in the *Registry* object passed to the *ConfiguratorContextBuilder*. If you use a class, it must have exactly one constructor, preferrably non-args. If you use args constructor, the arguments will be looked up from the Registry based on their types. If there are multiple beans applicable for one arguments, it will throw a *AmbigiousException*.

8.3 Using a custom Registry

Normally connectors will only require simple key-value configuration to work, but some might require specific **bean instances**. Bean are created and registered in `Registry`, where they can be *lookup* later. One example is the *MongoDBConnector* which requires an instance of *MongoClient*. To use those connectors, you will need to create the bean instances and register them inside the Registry.

```
// create the MongoClient bean
var mongoClient = MongoClient.create();

// register the bean
var myRegistry = new SimpleRegistry().register("mongoBean", mongoClient);

// create the context with the custom registry
var context = new DefaultGridgoContextBuilder().setName("application").
↪setRegistry(myRegistry).build();

// later on you can look up the bean using the provided name
mongoClient = context.getRegistry().lookup("mongoBean", MongoClient.class);
```

Another use case of Registry is to store some settings, which can be required for application to run. Since 0.2.0 you can also substitute the settings value in a connector endpoint.

Note: Since 0.2.0, you can register new entry after creating the Registry.

8.4 Bridge and Switch

Sometimes, you want a message coming from a Gateway's incoming sink to be automatically routed to another Gateway, then Bridge and Switch will come in handy. Some examples might be:

- Retrieve messages from Kafka, do some transformation and store it in Mongo

- Retrieve messages from incoming HTTP requests and routed it to a business logic Gateway

The different between Bridge and Switch is that Bridge will route the messages to the target Gateway's outgoing sink (using *send()*, while a Switch will route the messages to the target Gateway's incoming sink (using *push()*).

For example, the following code will use Bridge to store messages coming from Kafka to MongoDB:

```
context.openGateway("kafka")
       .attachConnector("kafka:someTopic?...");
context.openGateway("mongo")
       .attachConnector("mongo:mongoClient/test_db/test_collection?...");
context.attachComponent(new BridgeComponent("kafka", "mongo",␣
↪this::transformMessage));
```

Note that the *transformMessage()* method will convert the message from Kafka format to MongoDB format.

CHAPTER 9

Gridgo Boot Documentation

Gridgo Boot aims at providing a more convenient convention when working with Gridgo, using annotation instead of configuration.

9.1 Gridgo Boot Overview

Gridgo Boot is a framework to facilitate getting up and running with Gridgo. It provides an annotation-based approach, similar to Spring Boot to eliminate boilerplate code and tedious configurations.

9.1.1 Install

To install Gridgo Boot with Maven:

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-boot</artifactId>
    <version>${gridgo.version}</version>
</dependency>
```

9.1.2 Getting started

To start using Gridgo Boot, you need a Java main class.

```
@EnableComponentScan
@Registries(defaultProfile = "local")
public class Main {

    public static void main(String[] args) {
        GridgoApplication.run(Main.class, args);
```

(continues on next page)

(continued from previous page)

```
    }
}
```

`@EnableComponentScan` will instruct Gridgo Boot to scan for any Gridgo components and start instantiating it. These includes:

@Gateway Annotating that the class represents a Gateway, which can be attached with *@Connector*. If the class is an implementation of *Processor*, it will be subscribed to the gateway automatically too. This class will be scanned further for additional field annotations (e.g @RegistryInject, @ComponentInject...). Full documentation will be covered at the Dependency Injection section.

@DataAccessObject Annotating that the class represents a DAO. The class must be an interface. Full documentation will be covered at the Gridgo Data section.

@Component Annotating that the class represents a generic component. It will then be available for dependency injection using *@ComponentInject*. The class instance's properties can also be injected the same way as *@Gateway*.

9.2 Gridgo Boot Initializers

9.2.1 Registry Initializers

By default, Gridgo Boot will create a *MultiSourceRegistry* with the following child registries in that order:

- *SystemPropertyRegistry*: Using Java system properties.
- *SystemEnvRegistry*: Using system environment variables.
- Profile-specific *PropertiesFileRegistry*: Based on the *defaultProfile* settings or the Java system property *gridgo.profile* or the environment variable *gridgo_profile*.
- Application *PropertiesFileRegistry*: Using *application.properties*

If you want to register custom beans into the registry, add a public static method with *@RegistryInitializer* in the Main class:

```
@RegistryInitializer
public static void initRegistry(Registry registry) {
    // Register your custom beans here
    registry.register("someBean", new SomeBean());
}
```

If you want to add more registries, add a public static method with *@RegistryFactory* in the Main class:

```
@RegistryFactory
public static Registry createRegistry(Registry registry) {
    return new MyRegistry();
}
```

The added registries will take precedence over the default ones.

9.3 Gridgo Boot Dependency Injection

Gridgo Boot supports a simple built-in DI mechanism. First you create a class annotated with one of the following: *@Component*, *@Gateway*, *@DataAccessObject*. Then that class will be available to be injected into other classes using *@ComponentInject*, *@GatewayInject* and *@DataAccessInject*, respectively.

Examples:

1. Inject a @*Component* class:

```
@Component
class Foo {

}
```

Now inject Foo to other classes. Only classes annotated with @Component or @Gateway can be injected into

```
@Component
class Bar {

    @ComponentInject
    private Foo foo;
}

@Gateway("a_gateway")
@Connect("a_connector")
class SomeProcessor implements Processor {

    @ComponentInject
    private Foo foo;
}
```

2. Inject a @*Gateway* producer class

```
@Gateway("foo_gateway")
@Connect("a_connector_with_producer")
class SomeProducer {
}
```

Now inject the producer to another class. Only classes annotated with @Component or @Gateway can be injected into

```
@Gateway("bar_gateway")
@Connect("bar_connector")
class SomeProducer implements Processor {

    @GatewayInject("foo_gateway")
    private Gateway foo;
}
```

You may notice that in case of @*GatewayInject*, we need to specify the gateway name, and use *io.gridgo.core.Gateway* for the injected property instead of the actual gateway class. This is because it doesn't make any sense, since the class is usually an empty class anyway.

3. Inject a @DataAccessObject class

9.4 Gridgo Boot Tutorials

These are tutorials that might be helpful when working with Gridgo Boot

9.4.1 Using Prometheus instrumenters with Gridgo Boot

This tutorial will help you get up and running with Gridgo Boot Prometheus instrumenters. It is recommended that you read the instrumenters basic first. What it'll do:

1. Create a Prometheus instrumenter to measure a processor throughput and latency.
2. Expose Prometheus metrics through a HTTP endpoint.
3. Utilize the metrics in Grafana dashboard.

Installation

To start using Prometheus instrumenters, first you need to add it in your pom.xml

```
<dependency>
    <groupId>io.gridgo</groupId>
    <artifactId>gridgo-extras-prometheus</artifactId>
    <version>${gridgo.version}</version>
</dependency>
```

Create and register an instrumenter

The next step is to create and register the instrumenter in Gridgo Registry. This can be done using a *@RegistryInitializer* method, which is probably placed inside your initialization class (the one you pass to *GridgoApplication.run(...)*, usually Main class or Initializer)

```
@RegistryInitializer
public static void initRegistry(Registry registry) {
    // create a Prometheus summary instrumenter, referring to Prometheus␣
↪documentation for more
    var instrumenter = new PrometheusSummaryTimeInstrumenter("my_summary", "My Summary
↪");
    // register it in our registry
    registry.register("myInstrumenter", instrumenter);
}
```

Attach the registered instrumenter to the processor

Simply add a *@Instrumenter* to your processor class with name you registered earlier.

```
@Gateway("my_gateway")
@Connector("${my_connector_endpoint}")
@Instrumenter("myInstrumenter")
public class MyProcessor extends AbstractProcessor {

    @Override
    public void process(RoutingContext rc, GridgoContext gc) {
        // handle the message as usual
    }
}
```

Expose the metrics via a HTTP endpoint

Now everything is setup, but Prometheus is pull-based, so you need to expose metrics via a HTTP endpoint so that the Prometheus server can scrape it. The *MetricsProcessor* is already provided for your convenience. *my_metrics_endpoint* might be something like *vertx:http://0.0.0.0:8080/metrics*

```
@Gateway("metrics")
@Connector("${my_metrics_endpoint}")
public class MyMetricsProcessor extends MetricsProcessor {

    public MyMetricsProcessor() {
        super("my_metrics");
    }
}
```

When you run the application, it should be accessible by going to http://localhost:8080/metrics

Visualize the metrics

Ask your administrator or infrastructure team to scrape it using the endpoint you have specified earlier. Then you can query it use something like:

rate(my_metrics_my_summary_time_count[30s]): Return the average throughput of last 30 seconds *rate(my_metrics_my_summary_time_sum[30s]) / rate(my_metrics_my_summary_time_count[30s])*: Return the average latency of last 30 seconds

CHAPTER 10

Examples

Below are the list of examples with their descriptions:

gridgo-example-first-app: This application will start a web server at port 8080, listen for incoming HTTP requests and response with the same content

Link: https://github.com/gridgo/gridgo-examples/tree/master/gridgo-example-first-app

gridgo-example-mongodb-vertx This application will start a web server at port 8088, listen for incoming HTTP requests and response with a whole collection of MongoDB. It requires a MongoDB server running at port 27017

Link: https://github.com/gridgo/gridgo-examples/tree/master/gridgo-example-mongodb-vertx

gridgo-example-tik-tac-toe This application will create a simple tik-tac-toe game, with a websocket & HTTP gateway, and some support gateway to handle game logic.

Link: https://github.com/gridgo/gridgo-examples/tree/master/gridgo-example-tik-tac-toe

gridgo-example-kafka-consumer This application will create a simple kafka producer and consumer to send and receive a message.

Link: https://github.com/gridgo/gridgo-examples/tree/master/gridgo-example-kafka-consumer

gridgo-example-gridgoboot This application simulates a simple making an order flow at the restaurant, your order will be placed in a queue and then process asynchronously. You could request a meal by making a HTTP *POST* request to local webserver at port 8080. This example bases on *gridgoboot* feature.

Link: https://github.com/gridgo/gridgo-examples/tree/master/gridgo-example-gridgoboot