User Guide

The User Guide guide introduces Gensons main components and some features that can be enabled via configuration. After reading it you should be able to do address most of your use cases.

Overview

The main entry point in Genson library is the Genson class. It provides methods to serialize Java objects to JSON and deserialize JSON streams to Java objects. Instances of Genson are immutable and thread safe, you should reuse them. In general the recommended way is to have a single instance per configuration type. Genson provides a default constructor with no arguments that uses the default configuration. To customize Genson have a look at the GensonBuilder.

Read/Write Json

Genson by default supports main Java classes such as primitives, Lists, Maps, Dates, etc. by providing Converters for them, if there is no such converter Genson will consider it is a Java Bean and will try to ser/de based on its Getters, Setters, Fields and Constructors.

Once you have a Genson instance, you can use it to serialize/deserialize JSON strings.

```
int[] arrayOfInts = new int[]{1, 2, 3};
// json = [1,2,3]
String json = genson.serialize(arrayOfInts);
genson.deserialize(json, int[].class);
```

Supported IO classes

You can also use Java OutputStream/InputStream, Writer/Reader and byte arrays instead of Strings. In most cases you will want to use them instead of Strings for better performances.

```
byte[] byteArray = genson.serializeBytes(arrayOfInts);
genson.serialize(arrayOfInts, outputStream);
genson.serialize(arrayOfInts, writer);

genson.deserialize(byteArray, int[].class);
genson.deserialize(inputStream, int[].class);
genson.deserialize(reader, int[].class);
```

Untyped Java structures

During serialization if there is no type information available (meaning it is Object), Genson will use the runtime type of the object to decide how to serialize it. If it has type information then it will use it instead of the runtime type.

In the example below, Genson will use the runtime type of the values in the map.

```
Map<String, Object> person = new HashMap<String, Object>() {{
  put("name", "Foo");
  put("age", 28);
}};

// {"age":28,"name":"Foo"}
String json = genson.serialize(person);
```

Deserialization however is more tricky as the type must be detected based on the content of the JSON stream. By default, if no type information is available during deserialization, Genson will provide a base mapping between JSON types and Java types. For example, JSON object will be mapped to a Map with Strings as keys, JSON arrays to Java List, etc. There is another mechanism allowing to deserialize to polymorphic or unknown types, but we will discuss it latter.

Here in both cases it will be deserialized to a map with the correct types (string and int) as they are standard Json types.

```
Map<String, Object> map = genson.deserialize(json, Map.class);
Object map2 = genson.deserialize(json, Object.class);
```

Note: Sometime you might want to always use the runtime type, this can be enabled with

```
new GensonBuilder().useRuntimeType(true).create()
```

POJO Databinding

Of course working only with Maps, Lists and primitive types is not so convenient. Usually your goal is to map the JSON to some structure you have defined and vice versa.

Genson provides full support for object databinding by following standard JavaBean conventions. The basic rules for databinding are:

- All public or package visibility fields, getters and setters will be used, including the inherited ones.

- If a getter/setter exists for a field then it will be used instead of the field.
- Transient and static fields are not serialized nor used during deserialization.
- A default no argument constructor is required (can be configured to use constructors with arguments and support immutable types)
- If a field does not exist in the json stream then its field/setter will not be used.
- If a value is null then you can choose whether you want it to be serialized as null or just skipped, by default null values are serialized.

Lets have a closer look to an example.

```
Person someone = new Person("Eugen", 28, new Address(157, "Paris"));

// {"address":{"building":157,"city":"Paris"},"age":28,"name":"Eugen"}
String json = genson.serialize(someone);

// deserialize it back
Person person = genson.deserialize(json, Person.class);

public class Person {
  private String name;
  private int age;
  private Address address;

  public Person() {}

  public Person(String name, int age, Address address) {
    this.name = name;
    this.age = age;
    this.address = address;
  }

  // getters & setters
}

public class Address {
  public int building;
  public String city;

  public Address() {}

  public Address(int building, String city) {
    this.building = building;
    this.city = city;
  }
}
```

Remark Actually inner and anonymous classes serialization is supported but not deserialization, except if it is a static inner class.

You can also deserialize in an existing object. However this feature has some restrictions: it works only with Pojo like classes and does not handle nesting. For example if your root object contains a list or another Pojo, then it will be overriden by the one in the json (if present, otherwise it remains unchanged).

```
genson.deserializeInto(json, personInstance);
```

Filter/Rename properties

Genson provides two major ways of including/excluding and naming properties, via annotations and via configuration through GensonBuilder (not exclusive).

Via annotations

Excluding a property can be achieved by putting a **@JsonIgnore** annotation on the field or getter/setter. The same goes for including a property (for example a private field that you want to be serialized) but using **@JsonProperty** annotation. JsonProperty can also be used to define a different name for a property than the default one **@JsonProperty(**name**=**"newName"**)** .

Via GensonBuilder

The other way is using the GensonBuilder exclude/include/rename methods that support a wider range of exclusion/inclusion/naming patterns. This feature gets handy if you can't modify the source code of your classes or just don't want to pollute your code with annotations.

For example suppose you have an attribute named password in some of your classes and want to always exclude it or that you want to always exclude properties of some type, etc. The code below demonstrates some of those strategies (but more are available - see the API).

```
new GensonBuilder()
  .exclude("password")
  .exclude(Class.class)
  .include("name", FromSome.class)
  .rename("type", "_type")
.create();
```

Genson wide strategies

You can also define some strategies for property resolution that will be used for a Genson instance. You can choose whether to use Getter&Setter or Fields (or both - by default) and their

visibility (public, protected, private…).

```
new GensonBuilder()
  .useFields(true, VisibilityFilter.PRIVATE)
  .useMethods(false)
.create();
```

If you want to have a more general property resolution or naming strategy, you can respectively, extend PropertyBaseResolver or implement your own PropertyNameResolver and register it.

Object instantiation

By default Genson requires you to provide a default no argument constructor. But Genson also works with constructors that take arguments, encouraging people to work with immutable objects.

- *Why by default doesn't it support constructors with arguments?*
  Java reflection API does not provide access to parameter names of methods and constructors, Genson as most libs that deal with databinding uses it.

To bypass this problem Genson provides two solutions, annotations and automatic detection via byte code analysis.

Annotations

Annotate each parameter of the constructor with **@JsonProperty**, this would tell to Genson that this is the name that must be used.

```
public class Address {
  final int building;
  final String city;

  // we could also put @JsonCreator annotation, if we had multiple
  // constructors and wanted Genson to use this one.
  public Address(@JsonProperty("building") int building, @JsonProperty("city")
String city) {
    this.building = building;
    this.city = city;
  }
}
```

Automatic resolution

You can also use constructors with arguments without using annotations. This mechanism is disabled by default as it uses byte code parsing to extract names from code that has been compiled with debug symbols (the default for most build systems and libs). This option does not work on Android.

Enable automatic name resolution for constructors and methods via the GensonBuilder.

```
new GensonBuilder().useConstructorWithArguments(true).create();
```

Factory methods

Genson has also support for methods that act as factories. Those methods must be static, defined inside the class you want to deserialize (or a parent) and annotated with **@JsonCreator** annotation. JsonCreator annotation can also be used on Constructors when you have multiple ones and want to tell Genson which one to use.

In the previous example we could have a factory method that builds an Address with some default values, that would then be overridden by Genson based on the JSON content.

```
public class Address {
  final int building;
  final String city;

  @JsonCreator public static Address default() {
    return new Address(0, "UNKNOWN");
  }

  public Address(int building, String city) {
    this.building = building;
    this.city = city;
  }
}
```

Exceptions

Since version 0.99 there are no checked exceptions anymore. The main reason is to avoid ugly boilerplate code for people who use Genson. Of course even if Genson makes use of unchecked exceptions you can still catch them if you want to handle them. Genson can throw two kind of exceptions:

- **JsonStreamException**, thrown by the streaming API when parsing for example invalid JSON.
- **JsonBindingException**, everywhere else, especially if something goes wrong at the databinding level, ex: can not instantiate some type.

Generic types

Due to type erasure, in Java, we can not do things like List<Address>.class. Meaning that we can’t tell to Genson to deserialize to a List of Addresses but only to a list of unknown objects. Those unknown objects being deserialized as Maps. The solution is to use what is called TypeToken, described in this blog (http://gafter.blogspot.fr/2006/12/super-type-tokens.html). Gensons implementation of TypeToken is named GenericType. Why not TypeToken? Because

this name is used in Guava library so choosing another name allows to avoid confusion and mistakes.

```
List<Integer> listOfInt = genson.deserialize("[1, 2]", new
GenericType<List<Integer>>(){});
```

Genson has full support of Javas generics, allowing you to hava quite complex structures and still be able to ser/de it.

```
// enable constructor parameter names resolution through byte code analysis
Genson genson = new GensonBuilder().useConstructorWithArguments(true).create();

AddressGroup addressGroup = new AddressGroup(new Container<EuropeanAddress>(
    Arrays.asList(new EuropeanAddress("Champs Elysees", 1, "Paris")))
  );

// {"addressGroup":{"values":[{"building":1,"city":"Paris","street":"Champs
Elysees"}]}}
String json = genson.serialize(addressGroup);

public class EuropeanAddress extends Address {
  public final String street;

  public EuropeanAddress(String street, int building, String city) {
    super(building, city);
    this.street = street;
  }
}

public class AddressGroup {
  public final Container<EuropeanAddress> addressGroup;

  public AddressGroup(Container<EuropeanAddress> addressGroup) {
    this.addressGroup = addressGroup;
  }
}

public class Container<E extends Address> {
  public final List<E> values;

  public Container(List<E> values) {
    this.values = values;
  }
}
```

Remark that if in AddressGroup we were using Address type instead of the concrete type, the serialized json wouldn't contain the street property. Indeed it would have been ser/de using Address type, using the runtime type can be enable via **new GensonBuilder().**useRuntimeType**(true)** .

Polymorphic types

Another nice feature of Genson is its ability to deserialize an object serialized with Genson back to its concrete type. Lets now enable runtime type resolution and serialize a Container of EuropeanAddress and then deserialize it back.

```
// lets also enable runtime type usage
Genson genson = new GensonBuilder().useRuntimeType(true).create();

Container<Address> addressContainer = new Container<Address>(
    Arrays.asList(new EuropeanAddress("Champs Elysees", 1, "Paris")))
  );

// As we enabled runtime type resolution the street property will be serialized
// {"values":[{"building":1,"city":"Paris","street":"Champs Elysees"}]}
String json = genson.serialize(addressContainer);

// however when we deserialize Genson does not have enough type information.
// It will expand the content of the array to Address and not EuropeanAddress
Container<? extends Address> result = genson.deserialize(json, Container.class);
```

Of course in the previous example we could have deserialized to a **new** GenericType**<**Container**<**EuropeanAddress**>>()** , but if there were multiple implementations of Address or just mixed EuropeanAddress and Address instances, you won't get the correct result.

The solution to this problem is to store in the serialized json the information about the concrete type. In Genson it is implemented via a metadata mechanism that will serialize class name + package during serialization and use it when deserializing back.

```
// lets enable class metadata ser/de, allowing to deserialize back polymorphic
types
Genson genson = new GensonBuilder()
      .useClassMetadata(true)
      .useRuntimeType(true)
    .create();

// {"values":
[{"@class":"my.package.EuropeanAddress","building":1,"city":"Paris","street":"Champs
Elysees"}]}
AddressGroup result = genson.serialize(addressContainer);
```

Genson provides an alias mechanism for classes, allowing to use the alias instead of the full class name. It is a good practice to do so as it gives you the ability to rename your class or package without any impact on the json (especially useful if you store json in a persistent storage) and it is also safer from a security point of view.

```
// addAlias creates an alias between a logic name and a full class name, this also
enables class metadata mechanism
Genson genson = new GensonBuilder()
      .addAlias("europeanAddress", EuropeanAddress.class)
      .useRuntimeType(true)
    .create();

// {"values":
[{"@class":"europeanAddress","building":1,"city":"Paris","street":"Champs
Elysees"}]}
genson.serialize(addressContainer);
```

Remarks

- Class metadata and the overall metadata mechanism is available only for object types and not for arrays or litteral values. Metadata must always be the first name/value pairs in json objects.
- Genson does not serialize, nor use class metadata for standard types (lists, primitives, etc).
- If you define a custom Converter and do not want Genson to use class metadata for types supported by your Converter use @HandleClassMetadata annotation.

Streaming API

Genson has lower memory consumption than DOM based parsers, because Genson does not need to store the complete JSON document in memory. This low memory footprint is being achieved via the Streaming API. As your objects are being serialized, the JSON is directly written to the output. Same during deserialization, as the JSON arrives in the stream, Genson will parse it and map to your objects.

This is a valuable feature in real life applications, like web applications or web services, where a client is writing/reading data through the network. The network is the most common bottleneck, paralleling the work while data is being transferred increases the throughput.

Genson databinding uses the streaming API, but you can also directly use it without the databinding layer in case where you really need something extremely fast, **in most cases you don’t as it is already very efficient**.

The main classes of the Streaming API are ObjectWriter and ObjectReader, they are internally used by Genson to write/read JSON data. When implementing your own Converters to achieve custom serialization/deserialization, you will use this low level API. But in some cases you might also want to use them directly without any databinding. You can build them through Genson class.

```
ObjectWriter writer = genson.createWriter(outputStream);
  ObjectReader reader = genson.createReader(inputStream);
```

For example writing some object, containing an array and some basic properties can be easily done.

```
// {"name":"Foo Bar","age":45,"childrenAges":[1,2,3]}
writer.beginObject()
    .writeString("name", "Foo Bar")
    .writeNumber("age", 45)
    .writeName("childrenAges").beginArray()
      .writeValue(1)
      .writeValue(2)
      .writeValue(3)
    .endArray()
  .endObject();

/* when using directly the streaming api outside the databinding you are free to
flush the data when
you want allowing to reduce the memory foot print, when closing the writer
everything is flushed. */
writer.close();
```

Reading back the previous structure is a bit more of code, mainly because of Java verbosity. Of course you can also use Genson to deserialize your JSON to a map and then just extract your properties from there. If you use this API in a Converter, then you might want to use JSR 353 types with Genson (/Documentation/Extensions/#using-jsr-353-types-with-genson) for intermediary representation and then map it to your structure. This would be OK, as it will be applied only to the part handled by your Converter, all the rest would still use directly the Streaming API.

```
String name = null;
int age = 0;
List<Integer> childrenAges = null;

reader.beginObject();
while (reader.hasNext()) {
  /* return an enum corresponding to the type of the value being read (OBJECT,
ARRAY, NULL...),
  allows to implement generic parsing based on the json content */
  ValueType valueType = reader.next();

  if ("name".equals(reader.name())) {
      name = reader.valueAsString();
  } else if ("age".equals(reader.name())) {
      age = reader.valueAsInt();
  } else if ("childrenAges".equals(reader.name())) {
      childrenAges = readAnArray(reader);
  } else {
    /* skip this value, and recursively all its children if it is an object or an
array,
    and continue the parsing */
    reader.skipValue();
    // or throw an error and stop
    throw new IllegalStateException("Unknown property " + reader.name());
  }
}
reader.endObject();

private List<Integer> readAnArray(ObjectReader reader) {
  List<Integer> array = new ArrayList<Integer>();
  reader.beginArray();
  while(reader.hasNext()) {
      reader.next();
      array.add(reader.valueAsInt());
  }
  reader.endArray();
  return array;
}
```

Custom Ser/De

In some cases Genson may not serialize/deserialize a class the way you would like. First you

should try to have a look at the different options available to you, such as Genson configuration via the GensonBuilder or annotations.

Custom Converter

If nothing seems to help solve your problem, then you might want to implement your-self the way a specific type is being ser/de. This is achieved through implementing a custom Converter. A Converter is mainly an implementation of a Serializer and Deserializer for a specific type. If you don't need to provide a logic for serialization **and** deserialization, then you can implement only one of them.

Lets move the previous code in a datastructure and handle it in our Converter.

```
Genson genson = new GensonBuilder().withConverters(new PersonConverter()).create();

genson.deserialize("{\"name\":\"Foo Bar\",\"age\":45,\"childrenAges\":[1,2,3]}",
Person.class);

public class PersonConverter implements Converter<Person> {
  private GenericType<List<Integer>> typeOfChildrenAges = new
GenericType<List<Integer>>() {};

  public void serialize(Person person, ObjectWriter writer, Context ctx) throws
Exception {
    writer.beginObject();
    writer.writeString("name", person.getName())
          .writeNumber("age", person.getAge())
          .writeName("childrenAges");

    /* Delegate the serialization of a list of integer to Genson, you can pass
null here.
    Note that this will involve a lookup for the matching converter. */
    ctx.genson.serialize(person.getChildrenAges(), writer, ctx);

    writer.endObject();
  }

  /* You don't have to worry for the object being null here, if it is null Genson
will
  handle it for you. */
  public Person deserialize(ObjectReader reader, Context ctx) throws Exception {
    Person person = new Person();
    reader.beginObject();

    while (reader.hasNext()) {
      reader.next();
      if ("name".equals(reader.name())) {
          person.setName(reader.valueAsString());
      } else if ("age".equals(reader.name())) {
          person.setAge(reader.valueAsInt());
      } else if ("childrenAges".equals(reader.name())) {
          // here we replace the boilerplate code by delegating to genson
          person.setChildrenAges(ctx.genson.deserialize(typeOfChildrenAges,
reader, ctx));
      } else {
          reader.skipValue();
      }
    }

    reader.endObject();
```

```
    return person;
  }
}

public class Person {
  private String name;
  private int age;
  private List<Integer> childrenAges;

  // constructors & setters & getters
}
```

Like for the property childrenAges, in practice when implementing your own Converter, you will probably have to deal with more complex structures. Delegating to Genson the ser/de of complex types that you don't need to handle can greatly reduce the amount of code.

You can also create a custom Serializer and Deserializer if you want to handle only serialization or deserialization.

Converter Factory

Converters are registered for specific types. The one they have in their signature and additional ones that you can specify when registering the Converter.

```
new GensonBuilder().withConverter(personConverter, Person.class).create();
```

But in some cases you may want to be able to register a Converter for a complete hierarchy of types, ie. for Person and all its subclasses. This is one of the reasons Converter Factory has been introduced in Genson. The main role of a Factory is to return a Converter instance for a given type or null if the Factory does not handle this type.

```
new GensonBuilder().withConverterFactory(new PersonConverterFactory()).create();

public class PersonConverterFactory implements Factory<Converter<Person>> {

  /*Create method will be only called when type is Person or a subclass, you can
safely assume
  the upper bound of type is Person. The obtained Converter will be cached for
further reuse.*/
  public Converter<Person> create(Type type, Genson genson) {

    /*TypeUtil class provides methods to deal with java.lang.reflect.Type similar
to what you can
     find in TypeToken class of Guava. At the moment this class is more dedicated
to internal
     use, that's why the API lacks of documentation and is a bit rudimentary.*/
    Class<?> rawClass = TypeUtil.getRawClass(type);

    /*Instead of calling serialize method of the genson object in the Converter;
involving a
    lookup for the right Converter; you can do it only once in the Factory and
then provide it
    to your converter.*/
    Type typeOfListOfInt = new GenericType<List<Integer>>() {}.getType();
    Converter<List<Integer>> listOfIntConverter =
genson.provideConverter(typeOfListOfInt);

    return new PersonConverter(listOfIntConverter);
  }
}
```

Contextual Converter

Another nice feature in Genson is the support of Contextual Converters and Factories. The contextual converter is used through **@JsonConverter** annotation, that you put on object properties (fields, methods, constructor arguments). This will tell Genson to use this specific Converter for that property instead of any other.

You can also implement a ContextualFactory to create Converters based on the property

- type
- name
- annotations
- the class in which the property is declared.

This can become really handy if you want to apply some custom Converters based on such criteria. The **@JsonConverter** and **@JsonDateFormat** support have been implemented using Contextual Factories.

For example if you want to use some specific converter only when some custom annotation is present on the property:

```
public class MyContextualFactory implements ContextualFactory {
  @Override
  public Converter create(BeanProperty property, Genson genson) {
    MyCustomAnnotation ann = property.getAnnotation(MyCustomAnnotation.class);
    if (ann != null) {
      return myCustomConverter;
    } else return null;
  }
}

Genson genson = new GensonBuilder().withContextualFactory(new
MyContextualFactory()).create();
```

A couple of other things & components can be customized, but they are a even more advanced usage and require some knowledge of how Genson works, thus they are not documented here. If you encounter any problems or don't know how to achieve something, don't hesitate to post a message on the mailling list. You can also get help there on how to extend more "internal" parts of Genson.

(https://github.com/owlike/genson)