Type Coercion

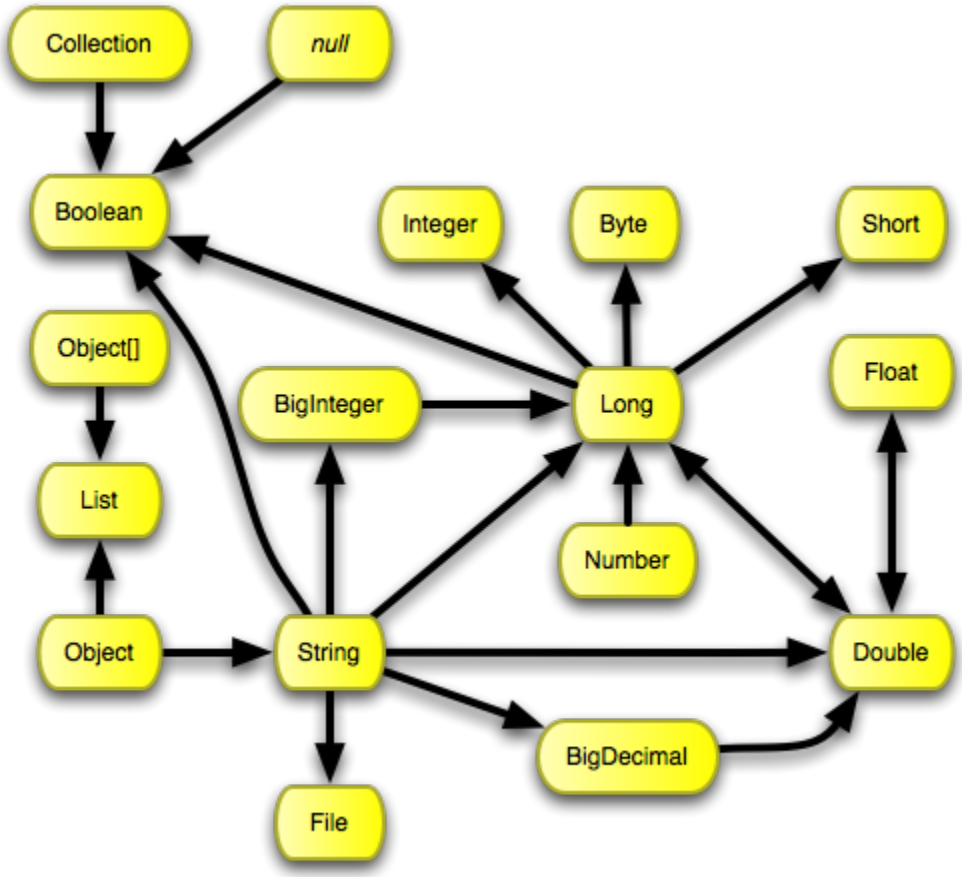
Type Coercion is the conversion of one type of object to a another object of a different type with similar content. Tapestry frequently must coerce objects from one type to another. A common example is the coercion of string "5" into an integer 5 or a double 5.0.

Although type coercions happen more inside tapestry-core (including coercions of component parameters), they may also occur inside tapestry-ioc, such as when injecting a value, rather than a service, into a builder method.

Like everything else in Tapestry, type coercions are extensible. At the root is the TypeCoercer service. Its configuration consists of CoercionTuples until 5.6.x and a mapped configuration of (CoercionTuple.Key, CoercionTuple) from Tapestry 5.7.0 on. Each tuple defines how to coerce from one type to another. The initial set of coercions is focused primarily on coercions between different numeric types:

Related Articles

- Type Coercion
- Parameter Type Coercion



Default Type Coercions

There are a few special coercions related to `null` there; `Object` --> `List` wraps a lone object as a singleton list, we then need `null` --> `List` to ensure that `null` stays `null` (rather than a singleton list whose lone element is a `null`).

Type Coercion Interpolation

Tapestry will try to *interpolate* necessary coercions. For example, say it is necessary to coerce a `StringBuffer` to an `Integer`; the TypeCoercer service will chain together a series of coercions:

- `Object` --> `String`
- `String` --> `Long`
- `Long` --> `Integer`

Because `Object --> String` is always available, this might lead to an unexpected interpolation between incompatible types due to multiple intermediate coercions. This can be easily prevented by providing an explicit CoercionTuple between the desired types.

Coercing from null

Coercing from `null` is special; it is not a spanning search as with the other types. Either there is a specific coercion from `null` to the desired type, or no coercion takes places (and the coerced value is `null`).

The only built-in `null` coercion is from `null` to `boolean` (which is always false).

List of Coercions

As of Tapestry version 5.7, the following coercions are available:

```
// Since Tapestry 5.1/5.2
Double --> Float
Float --> Double
Long --> Boolean
Long --> Byte
Long --> Double
Long --> Integer
Long --> Short
Number --> Long
Object --> Object[]
Object --> String
Object --> java.util.List
Object[] --> java.util.List
String --> Boolean
String --> Double
String --> Long
String --> java.io.File
String --> java.math.BigDecimal
String --> java.math.BigInteger
String --> java.text.DateFormat
String --> java.util.regex.Pattern
String --> org.apache.tapestry5.Renderable
String --> org.apache.tapestry5.SelectModel
String --> org.apache.tapestry5.corelib.ClientValidation
String --> org.apache.tapestry5.corelib.LoopFormState
String --> org.apache.tapestry5.corelib.SubmitMode
String --> org.apache.tapestry5.corelib.data.BlankOption
String --> org.apache.tapestry5.corelib.data.GridPagerPosition
String --> org.apache.tapestry5.corelib.data.InsertPosition
String --> org.apache.tapestry5.ioc.Resource
String --> org.apache.tapestry5.ioc.util.TimeInterval
boolean[] --> java.util.List
byte[] --> java.util.List
char[] --> java.util.List
double[] --> java.util.List
float[] --> java.util.List
int[] --> java.util.List
java.math.BigDecimal --> Double
java.util.Collection --> Boolean
java.util.Collection --> Object[]
java.util.Collection --> org.apache.tapestry5.grid.GridDataSource
java.util.Date --> java.util.Calendar
java.util.List --> org.apache.tapestry5.SelectModel
java.util.Map --> org.apache.tapestry5.SelectModel
long[] --> java.util.List
null --> Boolean
null --> org.apache.tapestry5.grid.GridDataSource
org.apache.tapestry5.ComponentResources --> org.apache.tapestry5.PropertyOverrides
org.apache.tapestry5.PrimaryKeyEncoder --> org.apache.tapestry5.ValueEncoder
org.apache.tapestry5.Renderable --> org.apache.tapestry5.Block
org.apache.tapestry5.Renderable --> org.apache.tapestry5.runtime.RenderCommand
org.apache.tapestry5.ioc.util.TimeInterval --> Long
org.apache.tapestry5.runtime.ComponentResourcesAware --> org.apache.tapestry5.ComponentResources
short[] --> java.util.List


// Since Tapestry 5.6.2

java.time.Year --> Integer
```

```
Integer        --> java.time.Year

java.time.Month --> Integer
Integer         --> java.time.Month
String          --> java.time.Month

java.time.YearMonth --> java.time.Year
java.time.YearMonth --> java.time.Month
String              --> java.time.YearMonth

java.time.MonthDay --> java.time.Month
String             --> java.time.MonthDay

java.time.DayOfWeek --> Integer
Integer             --> java.time.DayOfWeek
String              --> java.time.DayOfWeek

java.time.LocalDate --> java.time.Instant
java.time.Instant   --> java.time.LocalDate
String              --> java.time.LocalDate
java.time.LocalDate --> java.time.YearMonth
java.time.LocalDate --> java.time.MonthDay

java.time.LocalTime --> Long
Long                --> java.time.LocalTime
String              --> java.time.LocalTime

java.time.LocalDateTime --> java.time.Instant
java.time.Instant       --> java.time.LocalDateTime
String                  --> java.time.LocalDateTime
java.time.LocalDateTime --> jva.time.LocalDate

java.time.OffsetDateTime --> java.time.Instant
java.time.OffsetDateTime --> java.time.OffsetTime
String                   --> java.time.OffsetDateTime

String --> java.time.ZoneId

java.time.ZonedDateTime --> java.time.Instant
java.time.ZonedDateTime --> java.time.ZoneId
String                  --> java.time.ZonedDateTime

java.time.Instant --> Long
Long              --> java.time.Instant
java.time.Instant --> java.util.Date.class
java.util.Date    --> java.time.Instant

java.time.Duration --> Long
Long               --> java.time.Duration

String --> java.time.Period
```

Contributing New Coercions

TypeCoercer is extensible; you may add new coercions as desired. For example, let's say you have a `Money` type that represents an amount of some currency, and you want to be able to convert from `BigDecimal` to `Money`. Further, let's assume that `Money` has a constructor that accepts a `BigDecimal` as its parameter. We'll use a little Tapestry IOC configuration jujitsu to inform the TypeCoercer about this coercion.

AppModule (partial, Tapestry 5.7.0+)

```
public static void contributeTypeCoercer(MappedConfiguration<CoercionTuple.Key, CoercionTuple> configuration)
{
    Coercion<BigDecimal, Money> coercion = new Coercion<BigDecimal, Money>()
    {
        public Money coerce(BigDecimal input)
        {
           return new Money(input);
        }
    };
    CoercionTuple tuple = new CoercionTuple<BigDecimal, Money>(BigDecimal.class, Money.class, coercion);
    configuration.add(tuple.getKey(), tuple));
}
```

AppModule.java (partial, before Tapestry 5.7.0)

```
public static void contributeTypeCoercer(Configuration<CoercionTuple> configuration)
{
    Coercion<BigDecimal, Money> coercion = new Coercion<BigDecimal, Money>()
    {
        public Money coerce(BigDecimal input)
        {
           return new Money(input);
        }
    };

    configuration.add(new CoercionTuple<BigDecimal, Money>(BigDecimal.class, Money.class, coercion));
}
```

Further, since TypeCoercer knows how to convert `Double` to `BigDecimal`, or even `Integer` (to `Long` to `Double`) to `BigDecimal`, all of those coercions would work as well.

When creating a coercion from `null`, use `Void.class` as the source type. For example, the built-in coercion from `null` to `Boolean` is implemented as:

AppModule.java (partial)

```
    CoercionTuple tuple = new CoercionTuple(void.class, Boolean.class,
        new Coercion<Void, Boolean>()
    {
        public Boolean coerce(Void input)
        {
            return false;
        }
    }));
```

java.time (JSR310) Type Coercers

With Java 8, the Java Time API (JSR310) got added to the JDK, providing a new, more straightforward way of dealing with date and time. The multitude of different types created a need for additional TypeCoercers. But due to the way the Java Time API was implemented, you must be aware of some edge cases and intricacies.

Milliseconds Precision

Even though the Java Time API partially supports nanosecond precision, it's not guaranteed. Actually, the OS / the JVM implementation is responsible for providing the current time with `java.time.Clock`. Most of the new types even don't have convenience-methods for a nanosecond-based representation.

All date(time)-based types use `java.time.Instant` for coercion, which itself is coerced with milliseconds precision. This provides compatibility with `java.util.Date#getTime()`, which is also milliseconds-based.

Nanoseconds Precision

The 2 types not using milliseconds are not subclasses of or convertible to `java.time.Instant`:

- `java.time.LocalTime`: Uses nanoseconds internally, and no convenience method for milliseconds exists.
- `java.time.Duration`: Uses nanoseconds and seconds internally. Even though a convenience method for milliseconds exists, the type isn't dependent on the OS/JVM precision. So the highest possible precision is used.

Timezones

The Java Time API supports timezone-less types, e.g. `java.time.LocalDate`, `java.time.LocalDateTime`.
However, coercing these types into a `java.time.Instant` requires a timezone, because `java.time.Instant` uses the unix timestamp 0 as reference.

To still use automatic type coercion, the *Local* types will be seen as being in `ZoneId.systemDefault()`.

Additionally, `java.time.LocalDate` we use the local time of *00:00* for it's coercion to `java.time.Instant`.

Invalid Coercion Paths

Due to the powerful feature of finding compatible TypeCoercer paths to convert types with no explicit TypeCoercer, some additional TypeCoercers were added to ensure the correct conversion between types.

You should only try to coerce between the provided types and `java.time.Instant` as intermediate. Any other coercion path will require you to contribute a matching TypeCoercer yourself.