jtsgen Documentation

Release 0.5.0

Dragan Zuvic

Jun 15, 2021

Manual

The project *tsgen* (former *jtsgen*) emits TypeScript ambient types from Java sources. *tsgen* is implemented as an annotation processor, therefore it should be easily integrated in your current build infrastructure. Usually there are no other plugins required for your build system (maven, gradle).

CHAPTER 1

Introduction

The project *tsgen* converts Java types as TypeScript types at compile time using an annotation processor. The main use case of this project is getting code completion in the typescript frontend project for types that are defined in the JVM backend project. These backend types are usually written in Java (or Kotlin).

For example: the following class defines data type person, that represents the interface between the TypeScript based browser front end, e.g. angular or react, and a JAX-RS endpoint:

```
  public class Person {

      private String name;
      private LocalDate birthdate;
      private Sex sex;

      public Person(String name, LocalDate birthdate, Sex sex) {
          this.name = name;
          this.birthdate = birthdate;
          this.sex = sex;
      }

      public String getName() { return name; }

      public void setName(String name) { this.name = name; }

      public LocalDate getBirthdate() { return birthdate; }

     //...
}
```

The data might be transmitted and accessed using JSON, or whatever marshalling technology you like, but in TypeScript you have to write something like the following code for being able using code completion on that type. Either you write something like the following or just use this project, which generates the following data type:

```
export enum Sex {
  M, F
```

(continues on next page)

(continued from previous page)

```
}

export interface Person {
  birthdate: string;
  sex: Sex;
  name: string;
}
```

It is important to know, how the marshalling maps type like BigDecimal to JSON. For that the convesion of types could be easily configured by a DSL. And because this information how the Java types are mapped to JSON are configured somewhere in the backend, *tsgen* can also be configured at compile time, so the configuration of *tsgen* (package annotation) can be kept near the source the code that configured the Java to JSON mapper, e.g. Jackson's `ObjectMapper`.

1.1 Features

Currently the following features are supported:

- Emitting types for `@TypeScript` annotated Java classes and interfaces
- Ignoring a type using `@TSIgnore` annotation. Supporting `readonly` using `@TSReadOnly` annotation
- creating a module with corresponding package.json is supported. The name is constructed if not configured
- Configuration of the JavaScript / TypeScript module using the `@TSModule` annotation, e.g. the module name or the author of the exported ES module
- Configurable type conversion and exclusion using the `@TSModule` annotation. It also supports type parameters, e.g. a `ImmutableList<T>` can be be mapped to your own type `ImmutableArray<T>` with a corresponding (mapped) type `T`
- Java package as typescript name space, configurable
- converting getter/setter to TypeScript types
- Name Space mapping to minimize the TypeScript name spaces. It can be configured or calculated.
- Inheritance (since 0.2.0)
- Type variables without angle brackets in the generated TypeScript types, e.g. `T[]`
- Generics: type variables, type bounds and mapping to TypeScript types are supported

Requirements: The annotation processor only depends on the JDK. Only JDK 8 is currently supported. It *should* run on JDK 9.

This project is still in development, so major changes might occur and break when used, although it's used internally. Therefore either submit an issue on github or a pull request if you want a specific feature being implemented.

An example project ist also available on github: jtsgen-examples

1.2 Limits

As all annotation processors, only the types are transformed, no methods, or anything executable. For this you could try something like TeaVM or Kotlin JS.

Another limitation exists regarding name spaces until some major refactoring has been done: Currently the name space mapping is only available for java types, but not for type bounds. A type bound is converted without any name space. Usually that should be fine, unless you're planning extracting everything without any name space mapping.

This tool, like many other Java tools, is not able to infer missing types when acessing a Raw Type. This should usually not be problem, but please refer to F.A.Q. section, if this is an issue.

CHAPTER 2

Using jtsgen

The jtsgen annotation processor registers itself using the ServiceLoader protocol. Therefore when the processor is available on the compile or annotation class path it should be automatically invoked when compiling the annotated Java classes. Any Java class, interface or enum with the annotation `@TypeScript` will be converted, e.g.:

```
@TypeScript
public interface InterFaceSample {
    int getSomeInt();
    String getSomeString();
}
```

When compiling this class a complete ES module including a valid `package.json` is generated in the source output folder for a later deployment into a npm compatible repository.

Hint: Don't use the `jtsgen-processor` as a compile time or runtime dependency. Either get you build system to use the `javac` annotation class path or excluding it from the transitive dependencies, e.g. using `compileOnly` in Gradle or `optional` in Maven.

The generated sources are currently beneath the java source output folder. The output can be redirected using the regular `-s` option of javac.

2.1 Processing Parameters

The *tsgen* annotation processor supports with the following parameters:

- jtsgenLogLevel: enable additional logging. Use ine of the following `j.u.Logging` levels: `OFF` , `SEVERE` , `WARNING` , `INFO` , `CONFIG` , `FINE` , `FINER` , `FINEST` , `ALL`
- jtsgenModuleName: the name of the module, that should be exported
- jtsgenModuleVersion: the version number of the module
- jtsgenModuleDescription: the description of the module
- jtsgenModuleAuthor: the module author

- jtsgenModuleLicense: the npm license string of the module
- jtsgenModuleAuthorUr: the URL of the author

To use one of them, use the javac prefix `-A`, e.g.

```
javac -AjtsgenLogLevel=FINEST MyClass.java
```

All these settings do override everything set via an annotation, e.g. `TSModule`.

CHAPTER 3

Customizing Generation

The output generated by *tsgen* can be customized either by using the TSModule Annotation and/or by command line Parameters. The latter is useful embedding data from the build process, e.g. the version number.

The most relevant customizations are done adding one TSModule Annotation at a package of your compilation unit.

Sometimes a configuration can be placed multiple times, e.g. as compiler argument and as a setting in the TSModule annotation. The precedence of settings for the effective configuration is:

1. the default value in the TSModule annotation
2. as parameter of a used TSModule annotation
3. the arguments for the annotation processor

3.1 The TSModule Annotation

Currently only one TSModule annotation is permitted in one compilation unit. The annotation must be put to a package Element, like this:

```
@TSModule(
        moduleName = "namespace_test",
        nameSpaceMapping = "jts.modules.nsmap -> easy"
)
package jts.modules.nsmap;

import dz.jtsgen.annotations.TSModule;
```

The following annotation parameters are supported:

- **moduleName**: The module name of the JavaScript/TypeScript Module. This must be a java package friendly name. This is a required parameter, if the `TSModule` annotation is used
- **additionalTypes**: Array of full qualified Java type names, that should additionally be converted (since 0.4.0)
- **author**: The author number for the package.json file

- **authorUrl**: The authorURL for the package.json file
- **customTypeMappings**: Custom Type Mapping for the module, the default is `{}`
- **description**: the description for the package.json file
- **excludes**: regular expression to exclude type conversion, default is: `{"^sun", "^jdk.internal", "^java.lang.Comparable"}`
- **getterPrefixes**: The prefix filter for selecting the properties by getters. Default is: `{ "get([_a-zA-Z0-9].*)", "is([_a-zA-Z0-9]].*)"}`
- **generateTypeGuards**: Defines if typescript type gards should be generated as well. The default is false (since 0.3.0)
- **license**: The license for the package.json file
- **nameMappingStrategy**: The strategy for mapping getters / setters to member name: The default is `NameMappingStrategy.JACKSON_DEFAULT` (since 0.4.0)
- **nameSpaceMapping**: The name space mapping, the default is `{}`
- **nameSpaceMappingStrategy**: Defines how the default name space is calculated. Default is `NameSpaceMappingStrategy.ALL_TO_ROOT` (since 0.2.0)
- **outputType**: The type of the output. Default is `OutputType.NAMESPACE_AMBIENT_TYPE`
- **setterPrefixes**: prefix filter for members. Default is `{"set([_a-zA-Z0-9].*)"}` (sine 0.4.0)
- **version**: The version number for the package.json file, default is "1.0.0"
- **enumExportStrategy**: Defines how the default enum output strategy is. Default is `EnumExportStrategy.NUMERIC`

Note: The Processing Parameters *tsgen* may override some of these settings. See *Processing Parameters* for details.

3.2 Custom Type Mapping

The annotation processor supports a simple mapping description language. The custom Type Mapping for the module is a list of strings, each describing a type mapping. Each string consists of a Java Type (canonical name with type params) and the resulting TypeScript Type. Both Types are separated with an arrow, e.g. :

```
java.util.Date -> IDateJSStatic
```

maps a `java.util.Date` to the TypeScript type `IDateJSStatic`

It also is possible to use type variables, e.g. :

```
java.util.List<T> -> Array<T>
```

will convert any java.util.List or it's subtypes to an Array type in TypeScript. If the matched java type has a type parameter the converted type parameter will be inserted accordingly.

Because in TypeScript the types `Array` and `Map` differ from `[]` or `{}` jtsgen is able to embed the type in a literal way. After the arrow the type variables can be expresed using the back tick character, e.g:

```
java.util.List<T> -> `T`[]
```

Limits: There are some constraints using those expressions: it is not possible to express the name spaces at the right hand side in a proper way. jtsgen adds a namespace to the java declaration types. Currently accessing this name space is out of scope.

3.2.1 Default Conversions

The following mappings can not be configured, for now:

- The numerical primitive types are mapped to `number`
- The primitive boolean is mapped to `boolean`
- An array is mapped to 'T'[]

The annotation processor has the following mapping for declaration types configured:

- java.lang.Void -> Void
- java.lang.Object -> Object
- java.lang.String -> string
- java.lang.Character -> string
- java.lang.Number |-> number
- java.lang.Boolean -> boolean
- java.util.Collection<T> -> 'T'[]
- java.util.Map<U,V> -> { [key: 'U']: 'V'; }

The processor has no knowledge about the the necessary imports.

3.2.2 Mapping-DSL

The Mapping DSL defined in ANTLR BNF variant:

```
mapping : origin  whsp* arrow whsp* target;
arrow : '->' | '|->'
origin :  jident  ( '.' , jident )*   tsAngleType?
target :  ( jident  '.' )*  tstypes+
tsLit :  tsChar*
tsAngleType : '<'  jident  ( ','  jident )* '>'
tsLitType : '`'  jident  '`'
tsTypes : tstype | ( tstype whtsp )*
tsType :   tsLit | tsangletype | tslittype | whtsp

jident :  ('a'-'z' | 'A' - 'Z' | '_' )  ('a'-'z' | 'A' - 'Z' | '_'  |  '0' - '9')*
tsChar :  * all chars expecpt '<' | '>' | '`' *
```

3.3 Name Space Mapping

TSModule accepts a list of name spaces, that should me mapped (shortened). That list will be prepended to the calculated name space mapping. The following name spave mapping strategies are available:

- `TOP_LEVEL_TO_ROOT`: The top level java types are mapped to the root name space. Everything beneath is mapped into name spaces
- `ALL_TO_ROOT`: All types are mapped to the root name space, only the types of same name reside in their own name space
- `MANUAL`: No name space mapping is calculated

Some examples of :

- `a.b.c ->`: Maps a.b.c (and beneath) to root
- `a.b.c -> a.b`: Maps a.b.c to namespace a
- `=a.b.c ->`: Maps only a.b.c to the root

3.4 Output: TypeScript Modules

The type of the output can be configured by the outputType parameter of the TSModule annotation:

- *NAMESPACE_AMBIENT_TYPE* : exports a module with ambient types (d.ts and package.json) with a declared name space
- *NAMESPACE_FILE* : only the ambient types with namespaces in a single d.ts file
- *MODULE* : exports a declared module, e.g. using *declare module* at the top without ambient types
- *NO_MODULE* : exports a single file containing all converted types without any surrounding namespace or module declaration (since 0.2.0)

Unfortunately the TypeScript team decided to disable access to files outside of the rootDir [TS-9858]. The strategey including the output of *tsgen* into your frontend project depends on the general project structure. Use one of the following options:

1. Publish the generated module to the npmjs compatible repository (local or public). The disadvantage of this it that a an additional release step with a changed version number is needed for npm (or yarn) detecting a change
2. Using the *npm link* feature
3. No Module at all and instead copy the output directly into the TypeScript source directory.

3.5 Member Detection

tsgen detects the members of the converted type using the following rules:

1. public non static members of a Java class
2. existence of a *getter* method

By default *tsgen* adheres to the Java Beans specification [JavaBean], but this behavior can be modified. In the scope of this documentation the definition of *getter* and *setter* methods have to be extended to include members, that do not adhere the Java Beans specification, e.g. classes with Boolean properties generated by Kotlin. So:

1. a *setter* method is any method that returns void, accepts exactly one argument. The method name matches the defined *setter* expression.
2. a *getter* method is any method, that returns a type without any argument. The method name matches the defined *getter* expression.

To support the property naming conventions in Kotlin [CK] *tsgen* does not split getters to Boolean and non-Boolean types (isX, getX). For simplicity reasons only the following two options in `TSModule` define the *getter* and *setter* filter expressions:

- getterPrefixes default: `{ "get([_a-zA-Z0-9].*)", "is([_a-zA-Z0-9]].*)" }`
- setterPrefixes default: `{ "set([_a-zA-Z0-9]].*)"}`

Both prefixes act as an filter and as a function to extract the raw member of the member. The member name itself is defined by the member name mapping strategy (see next chapter). Only methods, that matches one of the both prefixes are considered as a *getter* or *setter* method. The group (that is the regular expression between the braces) extracts the the name which is applied to a name mapping function.

3.6 Member Name Mapping

The extracted raw member name has to match the one the used by the JSON framework. For example, in Jackson you define the mapping of the member names using a PropertyNamingStrategy [PNS]. *tsgen* tries to stick to the default setting of Jackson's DataBind. If necessary you can change this name mapping in *tsgen* by setting the `nameMappingStrategy` in `TSModule` to one of the following strategy:

- `JACKSON_DEFAULT`: the default Jackson property name mapping. This is the default used in *tsgen*
- `SIMPLE`: no mapping at all
- `SNAKE_CASE`: upper cases are interpreted as words that will be transformed to lower case words separated by underscores
- `UPPER_CAMEL_CASE`: The first character is converted to upper case

The member name mapping strategy can be defined using the parameter `nameMappingStrategy` of the `TSModule` annotation.

3.7 Enum Output Strategy

- `NUMERIC`: Writing numeric enums. This is the default used in *tsgen*
- `STRING`: Writing string enums.

CHAPTER 4

Using in Gradle projects

Adding the following snippet to your gradle (sub-) project, the annotation processor should run at automatically at compile time. Since 0.3.0 *tsgen* has been distributed on maven central, so no other repository has to be defined.

The used version should be held in a variable like the following

```
buildscript {
    ext {
      jtsgen_Version="0.4.0"
    }
 }
```

The brave ones could also try a development version, e.g. 0.5.0-SNAPSHOT. So the dependencies to jtsgen could be defined like the following. This should extract the TypeScript output to the gradle build dir automatically:

```
dependencies {
    compileOnly "com.github.dzuvic:jtsgen-annotations:${jtsgen_Version}"
    compileOnly "com.github.dzuvic:jtsgen-processor:${jtsgen_Version}"
}
```

Since Gradle 4.6 there is also a special dependency, if the compiler doesn't catch the processor automatically. So for Gradle 4.6+ it should look like this, by using the new dependency annotationProcessor :

```
dependencies {
    compileOnly "com.github.dzuvic:jtsgen-annotations:${jtsgen_Version}"
    compileOnly "com.github.dzuvic:jtsgen-processor:${jtsgen_Version}"
    annotationProcessor "com.github.dzuvic:jtsgen-processor:${jtsgen_Version}"
}
```

Please refer to the official Gradle Documentation around this subject.

4.1 Customizing the output dir

The output is customized by adding the source directory to the annotation processor:

```
def tsOutDir="${buildDir}/ts"

compileJava {
    options.compilerArgs = [ "-s", tsOutDir ]
    dependsOn(createTsDir)
}
```

4.2 Generating Types for Kotlin classes

If your project contains Kotlin classes that should be converted to Kotlin, you could try the kapt compiler plugin.

For example:

```
apply plugin: 'kotlin-kapt'

dependencies {

  compileOnly "com.github.dzuvic:jtsgen-annotations:${jtsgen_Version}"
  compileOnly "com.github.dzuvic:jtsgen-processor:${jtsgen_Version}"

  kapt "com.github.dzuvic:jtsgen-processor:${jtsgen_Version}"
 }

kapt {
  correctErrorTypes = true
  arguments {
      arg("jtsgenModuleVersion", version)
    }
}
```

CHAPTER 5

Using in Maven Projects

The annotation processor should run aromatically if `jtsgen-processor` is in the class path. Please note, that it is not advised making the processor transitive dependent in you project. Use either the `provided` or `optional` scope to break the dependency tree at this point. If it is necessary creating the TypesScript files in different location, the following strategies might solve this problem depending on the version of the maven-compiler plugin:

- for versions >= 2.2: maybe `annotationProcessors` could be useful
- for versions >= 2.2: with `generatedSourcesDirectory` the ouput path of the generated source files can be changed
- for versions >= 3.1: using `compilerArgs` the annotation processor can be fed with additional *Processing Parameters*
- for versions >= 3.5: with `annotationProcessorPaths` the processor could be completely removed from the dependency tree

For a full set of options you should take a look at the manual of the Maven Compiler Plugin.

The following full example generates the typescript files in `../client/src` and logs some info debug messages

```
<project>
    <modelVersion>4.0.0</modelVersion>
    <groupId>jtsgen-example</groupId>
    <artifactId>api</artifactId>
    <version>1.0.0-SNAPSHOT</version>

    <dependencies>
        <dependency>
            <groupId>com.github.dzuvic</groupId>
            <artifactId>jtsgen-processor</artifactId>
            <version>0.4.0</version>
            <scope>provided</scope>
        </dependency>
    </dependencies>

    <build>
```

(continues on next page)

(continued from previous page)

```
        <plugins>
            <plugin>
                <groupId>org.apache.maven.plugins</groupId>
                <artifactId>maven-compiler-plugin</artifactId>
                <version>3.3</version>
                <configuration>
                    <compilerVersion>1.8</compilerVersion>
                    <source>1.8</source>
                    <target>1.8</target>
                    <generatedSourcesDirectory>../client/src</
↪generatedSourcesDirectory>
                    <compilerArgs>
                        <arg>-AjtsgenLogLevel=info</arg>
                    </compilerArgs>
                </configuration>
            </plugin>
        </plugins>
    </build>
</project>
```

CHAPTER 6

Build and Develop

This chapter describes the internal structures of *tsgen* for development purpose. Also the build / release process and how to contribute to *tsgen*.

6.1 Build & Release

6.1.1 Build *tsgen*

To build *tsgen* you only need a JDK 8+ compatible environment on your machine, e.g.:

- AdoptOpenJDK – Available on https://adoptopenjdk.net
- Amazon Correto – Available on https://aws.amazon.com/corretto
- Azul OpenJDK 8+ – Available on https://www.azul.com/downloads/zulu
- IcedTea 3.8+ – Available on https://icedtea.classpath.org
- Oracle JDK 8+ – Available on https://java.com
- SapMachine – Available on https://sap.github.io/SapMachine

Tools like SdkMan are recommended for switching between various JDK versions.

Since version 0.5.0 the final releases have been built with IcdedTea.

Building on Linux

To build this project execute the following command:

```
$ ./gradlew build
Starting a Gradle Daemon, 1 incompatible and 1 stopped Daemons could not be reused,␣
↪use --status for details

```

(continues on next page)

(continued from previous page)

```
Deprecated Gradle features were used in this build, making it incompatible with␣
↪Gradle 5.0.
See https://docs.gradle.org/4.8/userguide/command_line_interface.html#sec:command_
↪line_warnings

BUILD SUCCESSFUL in 14s
18 actionable tasks: 10 executed, 8 up-to-date
```

6.2 Development

6.2.1 Structures

***tsgen* is an annotation processor, therefore it applies to an** implicit contract between the compiler and the processor. It only extracts the type information, mainly from the sources. Remember: some types are erased, especially on the classes

It processes the *Java* sources in the following stages:

1. The annotation processor `TsGenProcessor` is started by the compiler. It Analyzes only classes, that are annotated by the types from the `dz.jtsgen.annotations` package.
2. Before any Java conversion is processed, the a configuration structure is built by combining the command line arguments for the compiler and the information added to the `TSModule` annotation
3. After the configuration has been determined, the name space mappings are resolved.
4. The processor converts the Java types to an internal, AST like, structure for the render. There are multiple converters involved in this stage
5. It generates TypeScript code into the sources folder, which might be changed in the near future

```java
/*
 * Copyright (c) 2017 Dragan Zuvic
 *
 * This file is part of jtsgen.
 *
 * jtsgen is free software: you can redistribute it and/or modify
 * it under the terms of the GNU General Public License as published by
 * the Free Software Foundation, either version 3 of the License, or
 * (at your option) any later version.
 *
 * jtsgen is distributed in the hope that it will be useful,
 * but WITHOUT ANY WARRANTY; without even the implied warranty of
 * MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.  See the
 * GNU General Public License for more details.
 *
 * You should have received a copy of the GNU General Public License
 * along with jtsgen.  If not, see http://www.gnu.org/licenses/
 *
 */

package dz.jtsgen.processor.model;
import dz.jtsgen.processor.model.rendering.TSTypeElement;
import org.immutables.value.Value;

```

(continues on next page)

(continued from previous page)

```java
import javax.lang.model.element.Element;
import java.util.List;
import java.util.Optional;

import static dz.jtsgen.processor.util.StringUtils.lastOf;
import static dz.jtsgen.processor.util.StringUtils.untill;

/**
 * This type contains all information about a converted type
 */
public abstract class TSType implements TSTypeElement {


    @Value.Default
    public String getNamespace() {
        return untill(this.getElement().toString());
    }

    @Value.Default
    public String getName() {
        return lastOf(this.getElement().toString());
    }

    public abstract List<TSMember> getMembers();

    public abstract List<TSConstant> getConstants();

    public abstract Optional<String> getDocumentString();

    public abstract List<TSType> getSuperTypes();

    public abstract List<TSTypeVariable> getTypeParams();

    @Value.Parameter
    public abstract Element getElement();

    public abstract String getKeyword();

    public abstract TSType changedNamespace(String namespace, List<TSMember> members,␣
↪List<TSMethod> methods, List<TSConstant> mappedConstants);

    public abstract List<TSMethod> getMethods();
}
```

CHAPTER 7

Appendix

7.1 Frequently Asked Questions

This is a list of Frequently Asked Questions about *tsgen*.

... Is there a standalone version available?

tsgen is a kind of compiler plugin, therefore you need `javac` to generate TypeScript.

... `Option[T]` should be converted to an optional type?

TypeScript [TS] adds an `| undefined` to optional types if `--strictNullChecks` is enabled. So the following setting should to the same:

```
@TSModule(
     moduleName = "myModule",
     customTypeMappings = {
        "java.util.Optional[T] -> `T` | undefined"
        }
)
```

But be sure, that your JSON serializer does the same.

CHAPTER 8

References

CHAPTER 9

License And Legal Notes

No Warranty. As stated in the GPL v3 licence, there is no warranty of any kind by using this software.

The following licenses apply `jtsgen` / `tsgen`:

The **annotations** are **Apache 2.0** licensed. The **other parts** of `jtsgen` / `tsgen`, especially the processor, are **GPLv3** licensed. The license texts are included in the file `LICENSE`. Because `jtsgen` / `tsgen` as a sort of a compiler plugin you shouldn't redistribute the compiler in your projects. It's just like using OpenJDK: the generated code is *not* affected by it's license, so it should be safe using it in most cases. Everything in this chapter is not a legal advice in any form.

This project has to include the following legal notes:

- Oracle and Java are registered trademarks of Oracle and/or its affiliates. Other names may be trademarks of their respective owners. See https://www.oracle.com/legal/trademarks.html for details.

CHAPTER 10

Contributing

10.1 Contributors

In chronological order:

- Dragan Zuvic (dzuvic) Initial commit
- Frederik von Berg (fvonberg) Contributed some Bugfixes regarding generating boolean
- Markus Mangei (exonity) Added initial enumStrategy feature (#54)

CHAPTER 11

Change Log

The changelog is sorted descending by release date and contains at least one chapter about:

- *New features*
- *Breaking Changes*
- *Deprecated Features*, which will be removed in the next months
- *Removed Features*
- *Fixed*

11.1 jtsgen-0.6.0 (UNRELEASED)

11.1.1 Planned Features

- generating classes
- support for converting the documentation

11.2 jtsgen-0.5.0 <https://github.com/dzuvic/jtsgen/tree/jtsgen-0.5.0>‘__ (2021-03-13)

11.2.1 New Features

- The documentation has been expanded with a section about the contributors, building and extending `tsgen`
- Tests for JDK 11 have been added
- new options for generating enum
- enum values are rendered each in a new line

- documentation strings are rendered, but currently without any transformation to tsdoc
- add export of java methods to typescript methods using @TSMethod annotation
- add export of java constructors to typescript constructor functions using @TSConstructor annotation
- add export of functional interfaces as function types
- allow to rename exported methods/properties/types using the `name` property in annotations
- new option to add a timestamp to the module comment
- sort all kinds of outputs to simplify comparison of different versions
- add support for nullable values: Property members will can be undefined, methods can return null
- allow to add static imports in modules

11.2.2 Internal

- Migrated to Gradle 6.8

11.2.3 Tickets

Implemented enhancements: - String enums as alternative to Numeric enums

#56

- Emit Java documentation #4

Fixed bugs:

- java.lang.Boolean is not supported by default #52
- Using JDK9 some tests fail #44

11.3 jtsgen-0.4.0 (2018-05-06)

11.3.1 New Features

- Added documentation for Maven and Kotlin projects
- Added Support for type bounds on classes and interfaces
- Added support converting types without an annotation
- Added support for non standard properties
- Added support for different name mapping strategies

11.3.2 Breaking Changes

Type bounds of classes will be converted, but i don't expect that this shouldn't break any existing code.

11.3.3 Tickets

Implemented enhancements:

- publish snapshot #48
- Kotlin Boolean doesn't show up #42
- Bug: Generics in Ouput are missing #33

Fixed bugs:

- Generating bogus Java types due to not aborting the converter recursion #46
- Bug: wrong dependency in Processor #40

Closed issues:

- Support generating Ouptput without an annotation #41
- Add documentaion for maven #38
- Support for converting external types #36
- rename project packages prior deployment on maven central #12

Merged pull requests:

- Feature/46 early dsl #51 (dzuvic)
- Feature/48 publish snapshot #49 (dzuvic)
- Feature/33 generics #47 (dzuvic)

11.4 jtsgen-0.3.0 (2017-10-30)

11.4.1 New Features

- `@TSReadOnly` annotated members are exported as `readonly`
- support for literal mapping of types, e.g. `Array<T>` can be mapped to `T[]`
- migrated documentation from markdown to sphinx
- migrated from bintray to maven central

11.4.2 Breaking Changes

- The default mechanism that generates `readonly` when only getters are visible has been removed. Use the `@TSReadonly` annotation to generate readonly members
- The default conversion of collections and maps are changed to `T[]` and `{ index:  K: V; }`
- the artifact are distributed on maven central. Removing the custom repository used till 0.2.x releases should be sufficient

11.4.3 Tickets

Implemented enhancements:

- Bug: Java Bean Protocol not complete #32
- Map Collection<T> to T[] instead of List<T> #28
- Capability generate User-Defined Type Guards (enable basic TypeCheck at Runtime) #27

Fixed bugs:

- NPE when Type Mapping is not available #34

Closed issues:

- Make it available in maven central #37
- iInheritance: Only add member, when not in super types #30
- TSOption or TSReadOnly Annotation #17

Merged pull requests:

- [add] support for boolean class attributes #33 #35 (fvonberg)

11.5 jtsgen-0.2.0 (2017-07-14)

11.5.1 New Features

- Support for inheritance added
- Selectable name space mapping strategy
- Output file without any module or name space declaration

11.5.2 Breaking Change

- The default name space mapping changes to “ALL_ROOT”
- Defining a name space mapping doe not replace the calculated any more
- Renamed the OutputType enum members

11.5.3 Tickets

Implemented enhancements:

- change default name space mapping in order to avoid name spaces at all #26
- Missing “NO_MODULE” OutputType #25
- Please support inheritance #23
- support exporting for direct usage #15
- Support for no name space mapping #29

11.6 jtsgen-0.1.4 (2017-05-31)

Full Changelog

Implemented enhancements:

- support exporting for direct usage #15

11.7 jtsgen-0.1.3 (2017-05-27)

Full Changelog

Implemented enhancements:

- export java.lang.Object to Object instead of any #21

Fixed bugs:

- enum not used, but converted, when namespac mapping removes the package #19
- java.lang.Number -> number is not conveted #18
- remove jtsgen directory in the output. only use the modulename as directory #14

Closed issues:

- enable coverage using jacoco #22

11.8 jtsgen-0.1.2 (2017-05-15)

Full Changelog

Implemented enhancements:

- support exporting only the d.ts file #16

Fixed bugs:

- compile time dependency to guava #13

11.9 jtsgen-0.1.1 (2017-05-13)

Full Changelog

Implemented enhancements:

- name space mapping #10

11.10 jtsgen-0.1.0 (2017-05-10)

Full Changelog

Implemented enhancements:

- recursive type conversion of embedded types #11

- Support for Generics and nesting Types #8
- Support for Enums #6

Fixed bugs:

- name space generation missing last character #9

11.11 jtsgen-0.0.2 (2017-04-26)

Implemented enhancements:

- support for visibility of types and class attributes #5
- Add support for ignoring part of the Java Type #3
- User defined conversions #2

Closed issues:

- publish jtsgen to a public repo #1

* *This Change Log was automatically generated by ‘github_changelog_generator <https://github.com/skywinder/Github-Changelog-Generator>‘__*

CHAPTER 12

Indices and tables

- genindex
- modindex
- search

Bibliography

[JavaBean] Java Bean Specification <http://download.oracle.com/otndocs/jcp/7224-javabeans-1.01-fr-spec-oth-JSpec>

[CK] "Calling Kotlin from Java" from the Kotlin Reference <https://kotlinlang.org/docs/reference/java-to-kotlin-interop.html>

[PNS] https://fasterxml.github.io/jackson-databind/javadoc/2.9/com/fasterxml/jackson/databind/PropertyNamingStrategy.html

[TS] TypeScript <https://www.typescriptlang.org/docs>

[TS-9858] TypeScript issue regarding files outside `rootDir` <https://github.com/Microsoft/TypeScript/issues/9858>