Gelatin Documentation

Release 0.3.4

Samuel Abels

Jun 13, 2023

Contents

Gelatin turns your text soup into something solid. It is a combined lexer, parser, and output generator. It converts text files to XML, JSON, or YAML. It is a simple language for converting text into a structured formats.

CHAPTER 1

Development

Gelatin is on GitHub.

CHAPTER 2

License

Gelatin is published under the MIT licence.

CHAPTER 3

Contents

3.1 Quick Start

Suppose you want to convert the following text file to XML:

```
User
----
Name: John, Lastname: Doe
Office: 1st Ave
Birth date: 1978-01-01

User
----
Name: Jane, Lastname: Foo
Office: 2nd Ave
Birth date: 1970-01-01
```

The following Gelatin syntax does the job:

```
# Define commonly used data types. This is optional, but
# makes your life a litte easier by allowing to reuse regular
# expressions in the grammar.
define nl /[\r\n]/
define ws /\s+/
define fieldname /[\w ]+/
define value /[^\r\n,]+/
define field_end /[\r\n,] */

grammar user:
    match 'Name:' ws value field_end:
        out.add_attribute('.', 'firstname', '$2')
    match 'Lastname:' ws value field_end:
        out.add_attribute('.', 'lastname', '$2')
    match fieldname ':' ws value field_end:
```

(continues on next page)

(continued from previous page)

```
        out.add('$0', '$3')
    match nl:
        do.return()

# The grammar named "input" is the entry point for the converter.
grammar input:
    match 'User' nl '----' nl:
        out.open('user')
    user()
```

Explanation:

1. **"grammar input:"** is the entry point for the converter.
2. **"match"** statements in each grammar are executed sequentially. If a match is found, the indented statements in the match block are executed. After reaching the end of a match block, the grammar restarts at the top of the grammar block.
3. If the end of a grammar is reached before the end of the input document was reached, an error is raised.
4. **out.add('$0', '$3')** creates a node in the XML (or JSON, or YAML) if it does not yet exist. The name of the node is the value of the first matched field (the fieldname, in this case). The data of the node is the value of the fourth matched field.
5. **out.open('user')** creates a "user" node in the output and selects it such that all following "add" statements generate output relative to the "user" node. Gelatin leaves the user node upon reaching the out.leave() statement.
6. **user()** calls the grammar named "user".

This produces the following output:

```
<xml>
  <user lastname="Doe" firstname="John">
    <office>1st Ave</office>
    <birth-date>1978-01-01</birth-date>
  </user>
  <user lastname="Foo" firstname="Jane">
    <office>2nd Ave</office>
    <birth-date>1970-01-01</birth-date>
  </user>
</xml>
```

3.1.1 Starting the transformation

The following command converts the input to XML:

```
gel -s mysyntax.gel input.txt
```

The same for JSON or YAML:

```
gel -s mysyntax.gel -f json input.txt
gel -s mysyntax.gel -f yaml input.txt
```

3.1.2 Using Gelatin as a Python Module

Gelatin also provides a Python API for transforming the text:

```
from Gelatin import generator
from Gelatin.util import compile

# Parse your .gel file.
syntax = compile('syntax.gel')

# Convert your input file to XML.
builder = generator.Xml()
syntax.parse('input.txt', builder)
print builder.serialize()
```

3.2 Common Operations

3.2.1 Generating XML attributes

There are two ways for creating an attribute. The first is using URL notation within a node name:

```
grammar input:
    match 'User' nl '----' nl 'Name:' ws value field_end:
        out.enter('user?name="$6"')
        user()
```

The second, equivalent way calls add_attribute() explicitely:

```
grammar input:
    match 'User' nl '----' nl 'Name:' ws value field_end:
        out.enter('user')
        out.add_attribute('.', 'name', '$6')
        user()
```

3.2.2 Skipping Values

```
match /# .*[\r\n]/:
    do.skip()
```

3.2.3 Matching Multiple Values

```
match /# .*[\r\n]/
    | '/*' /[^\r\n]/ '*/' nl:
    do.skip()
```

3.2.4 Grammar Inheritance

A grammar that uses inheritance executes the inherited match statements before trying it's own:

```
grammar default:
    match nl:
        do.return()
```

(continues on next page)

(continued from previous page)

```
    match ws:
        do.next()

grammar user(default):
    match fieldname ':' ws value field_end:
        out.add('$0', '$3')
```

In this case, the *user* grammar inherits the whitespace rules from the *default* grammar.

3.3 Gelatin Syntax

The following functions and types may be used within a Gelatin syntax.

3.3.1 Types

STRING

A string is any series of characters, delimited by the ' character. Escaping is done using the backslash character. Examples:

```
'test me'
'test \'escaped\' strings'
```

VARNAME

VARNAMEs are variable names. They may contain the following set of characters:

```
[a-z0-9_]
```

NODE

The output that is generated by Gelatin is represented by a tree consisting of nodes. The NODE type is used to describe a single node in a tree. It is a URL notated string consisting of the node name, optionally followed by attributes. Examples for NODE include:

```
.
element
element?attribute1="foo"
element?attribute1="foo"&attribute2="foo"
```

PATH

A PATH addresses a node in the tree. Addressing is relative to the currently selected node. A PATH is a string with the following syntax:

```
NODE[/NODE[/NODE]...]'
```

Examples:

```
.
./child
parent/element?attribute="foo"
parent/child1?name="foo"/child?attribute="foobar"
```

REGEX

This type describes a Python regular expression. The expression MUST NOT extract any subgroups. In other words, when using bracket expressions, always use `(?:)`. Example:

```
/^(test|foo|bar)$/          # invalid!
/^(?:test|foo|bar)$/        # valid
```

If you are trying to extract a substring, use a match statement with multiple fields instead.

3.3.2 Statements

define VARNAME STRING|REGEX|VARNAME

define statements assign a value to a variable. Examples:

```
define my_test /(?:foo|bar)/
define my_test2 'foobar'
define my_test3 my_test2
```

match STRING|REGEX|VARNAME ...

Match statements are lists of tokens that are applied against the current input document. They parse the input stream by matching at the current position. On a match, the matching string is consumed from the input such that the next match statement may be applied. In other words, the current position in the document is advanced only when a match is found.

A match statement must be followed by an indented block. In this block, each matching token may be accessed using the *$X* variables, where *X* is the number of the match, starting with *$0*.

Examples:

```
define digit /[0-9]/
define number /[0-9]+/

grammar input:
    match 'foobar':
        do.say('Match was: $0!')
    match 'foo' 'bar' /[\r\n]/:
        do.say('Match was: $0!')
    match 'foobar' digit /\s+/ number /[\r\n]/:
        do.say('Matches: $1 and $3')
```

You may also use multiple matches resulting in a logical OR:

```
match 'foo' '[0-9]' /[\r\n]/
    | 'bar' /[a-z]/ /[\r\n]/
```

(continues on next page)

(continued from previous page)

```
    | 'foobar' /[A-Z]/ /[\r\n]/:
    do.say('Match was: $1!')
```

imatch STRING|REGEX|VARNAME ...

imatch statements are like *match* statements, except that matching is case-insensitive.

when STRING|REGEX|VARNAME ...

when statements are like *match* statements, with the difference that upon a match, the string is not consumed from the input stream. In other words, the current position in the document is not advanced, even when a match is found. *when* statements are generally used in places where you want to "bail out" of a grammar without consuming the token.

Example:

```
grammar user:
    match 'Name:' /\s+/ /\S+/ /\n/:
        do.say('Name was: $2!')
    when 'User:':
        do.return()

grammar input:
    match 'User:' /\s+/ /\S+/ /\n/:
        out.enter('user/name', '$2')
        user()
```

skip STRING|REGEX|VARNAME

skip statements are like *match* statements without any actions. They also do not support lists of tokens, but only one single expression.

Example:

```
grammar user:
    skip /#.*?[\r\n]+/
    match 'Name: ' /\s+/ /\n/:
        do.say('Name was: $2!')
    when 'User:':
        do.return()
```

3.3.3 Output Generating Functions

out.create(PATH[, STRING])

Creates the leaf node (and attributes) in the given path, regardless of whether or not it already exists. In other words, using this function twice will lead to duplicates. If the given path contains multiple elements, the parent nodes are only created if the do not yet exist. If the STRING argument is given, the new node is also assigned the string as data. In other words, the following function call:

```
out.create('parent/child?name="test"', 'hello world')
```

leads to the following XML output:

```
<parent>
    <child name="test">hello world</child>
</parent>
```

Using the same call again, like so:

```
out.create('parent/child?name="test"', 'hello world')
out.create('parent/child?name="test"', 'hello world')
```

the resulting XML would look like this:

```
<parent>
    <child name="test">hello world</child>
    <child name="test">hello world</child>
</parent>
```

out.replace(PATH[, STRING])

Like out.create(), but replaces the nodes in the given path if they already exist.

out.add(PATH[, STRING])

Like out.create(), but appends the string to the text of the existing node if it already exists.

out.add_attribute(PATH, NAME, STRING)

Adds the attribute with the given name and value to the node with the given path.

out.open(PATH[, STRING])

Like out.create(), but also selects the addressed node, such that the PATH of all subsequent function calls is relative to the selected node until the end of the match block is reached.

out.enter(PATH[, STRING])

Like out.open(), but only creates the nodes in the given path if they do not already exist.

out.enqueue_before(REGEX, PATH[, STRING])

Like out.add(), but is not immediately executed. Instead, it is executed as soon as the given regular expression matches the input, regardless of the grammar in which the match occurs.

out.enqueue_after(REGEX, PATH[, STRING])

Like out.enqueue_before(), but is executed after the given regular expression matches the input and the next match statement was processed.

out.enqueue_on_add(REGEX, PATH[, STRING])

Like out.enqueue_before(), but is executed after the given regular expression matches the input and the next node is added to the output.

out.clear_queue()

Removes any items from the queue that were previously queued using the out.enqueue_*() functions.

3.3.4 Control Functions

do.skip()

Skip the current match and jump back to the top of the current grammar block.

do.next()

Skip the current match and continue with the next match statement without jumping back to the top of the current grammar block.
This function is rarely used and probably not what you want. Instead, use do.skip() in almost all cases, unless it is for some performance-specific hacks.

do.return()

Immediately leave the current grammar block and return to the calling function. When used at the top level (i.e. in the *input* grammar), stop parsing.

do.say(STRING)

Prints the given string to stdout, with additional debug information.

do.fail(STRING)

Like do.say(), but immediately terminates with an error.

3.4 Python API

3.4.1 Gelatin.util module

Gelatin.util.**compile**(*syntax_file*, *encoding='utf8'*)

Like compile_string(), but reads the syntax from the file with the given name.

Parameters

- **syntax_file** (*str*) – Name of a file containing Gelatin syntax.
- **encoding** (*str*) – Character encoding of the syntax file.

Return type compiler.Context

Returns The compiled converter.

Gelatin.util.**compile_string**(*syntax*)
Builds a converter from the given syntax and returns it.

Parameters **syntax** (*str*) – A Gelatin syntax.

Return type compiler.Context

Returns The compiled converter.

Gelatin.util.**generate**(*converter*, *input_file*, *format='xml'*, *encoding='utf8'*)
Given a converter (as returned by compile()), this function reads the given input file and converts it to the requested output format.

Supported output formats are 'xml', 'yaml', 'json', or 'none'.

Parameters

- **converter** (*compiler.Context*) – The compiled converter.
- **input_file** (*str*) – Name of a file to convert.
- **format** (*str*) – The output format.
- **encoding** (*str*) – Character encoding of the input file.

Return type str

Returns The resulting output.

Gelatin.util.**generate_string**(*converter*, *input*, *format='xml'*)
Like generate(), but reads the input from a string instead of from a file.

Parameters

- **converter** (*compiler.Context*) – The compiled converter.
- **input** (*str*) – The string to convert.
- **format** (*str*) – The output format.

Return type str

Returns The resulting output.

Gelatin.util.**generate_string_to_file**(*converter*, *input*, *output_file*, *format='xml'*, *out_encoding='utf8'*)
Like generate(), but reads the input from a string instead of from a file, and writes the output to the given output file.

Parameters

- **converter** (*compiler.Context*) – The compiled converter.
- **input** (*str*) – The string to convert.
- **output_file** (*str*) – The output filename.
- **format** (*str*) – The output format.
- **out_encoding** (*str*) – Character encoding of the output file.

Return type str

Returns The resulting output.

Gelatin.util.**generate_to_file**(*converter*, *input_file*, *output_file*, *format='xml'*, *in_encoding='utf8'*, *out_encoding='utf8'*)

Like generate(), but writes the output to the given output file instead.

Parameters

- **converter** (*compiler.Context*) – The compiled converter.
- **input_file** (*str*) – Name of a file to convert.
- **output_file** (*str*) – The output filename.
- **format** (*str*) – The output format.
- **in_encoding** (*str*) – Character encoding of the input file.
- **out_encoding** (*str*) – Character encoding of the output file.

Return type str

Returns The resulting output.

3.4.2 Gelatin.compiler package

Module contents

Gelatin.compiler.Context module

class Gelatin.compiler.Context.**Context**

Bases: object

__init__()

x.__init__(...) initializes x; see help(type(x)) for signature

dump()

parse(*filename*, *builder*, *encoding='utf8'*, *debug=0*)

parse_string(*input*, *builder*, *debug=0*)

Gelatin.compiler.Context.**do_fail**(*context*, *message='No matching statement found'*)

Gelatin.compiler.Context.**do_next**(*context*)

Gelatin.compiler.Context.**do_return**(*context*, *levels=1*)

Gelatin.compiler.Context.**do_say**(*context*, *message*)

Gelatin.compiler.Context.**do_skip**(*context*)

Gelatin.compiler.Context.**do_warn**(*context*, *message*)

Gelatin.compiler.Context.**out_add**(*context*, *path*, *data=None*)

Gelatin.compiler.Context.**out_add_attribute**(*context*, *path*, *name*, *value*)

Gelatin.compiler.Context.**out_clear_queue**(*context*)

Gelatin.compiler.Context.**out_create**(*context*, *path*, *data=None*)

Gelatin.compiler.Context.**out_enqueue_after**(*context*, *regex*, *path*, *data=None*)

Gelatin.compiler.Context.**out_enqueue_before**(*context*, *regex*, *path*, *data=None*)

Gelatin.compiler.Context.**out_enqueue_on_add**(*context*, *regex*, *path*, *data=None*)

Gelatin.compiler.Context.**out_enter**(*context*, *path*)

Gelatin.compiler.Context.**out_open**(*context*, *path*)

Gelatin.compiler.Context.**out_replace**(*context*, *path*, *data=None*)

Gelatin.compiler.SyntaxCompiler module

class Gelatin.compiler.SyntaxCompiler.**SyntaxCompiler**
Bases: simpleparse.dispatchprocessor.DispatchProcessor

Processor sub-class defining processing functions for the productions.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

define_stmt(*token*, *buffer*)

grammar_stmt(*token*, *buffer*)

reset()

3.4.3 Gelatin.generator package

Module contents

Gelatin.generator.**new**(*format*)

Gelatin.generator.Builder module

class Gelatin.generator.Builder.**Builder**
Bases: object

Abstract base class for all generators.

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

add(*path*, *data=None*, *replace=False*)
Creates the given node if it does not exist. Returns the (new or existing) node.

add_attribute(*path*, *name*, *value*)
Creates the given attribute and sets it to the given value. Returns the (new or existing) node to which the attribute was added.

create(*path*, *data=None*)
Creates the given node, regardless of whether or not it already exists. Returns the new node.

dump()

enter(*path*)
Enters the given node. Creates it if it does not exist. Returns the node.

leave()
Returns to the node that was selected before the last call to enter(). The history is a stack, to the method may be called multiple times.

open(*path*)
Creates and enters the given node, regardless of whether it already exists. Returns the new node.

serialize(*serializer*)

serialize_to_file(*filename*)

class Gelatin.generator.Builder.**Node**(*name*, *attribs=None*)
Bases: object

__init__(*name*, *attribs=None*)
x.__init__(...) initializes x; see help(type(x)) for signature

add(*child*)

dump(*indent=0*)

get_child(*name*, *attribs=None*)
Returns the first child that matches the given name and attributes.

to_dict()

class Gelatin.generator.Builder.**OrderedDefaultDict**(*default_factory=None*, **a*, ***kw*)
Bases: collections.OrderedDict

__init__(*default_factory=None*, **a*, ***kw*)
Initialize an ordered dictionary. The signature is the same as regular dictionaries, but keyword arguments are not recommended because their insertion order is arbitrary.

copy() → a shallow copy of od

Gelatin.generator.Builder.**nodehash**(*name*, *attribs*)

Gelatin.generator.Dummy module

class Gelatin.generator.Dummy.**Dummy**
Bases: *Gelatin.generator.Builder.Builder*

__init__()
x.__init__(...) initializes x; see help(type(x)) for signature

add(*path*, *data=None*, *replace=False*)
Creates the given node if it does not exist. Returns the (new or existing) node.

add_attribute(*path*, *name*, *value*)
Creates the given attribute and sets it to the given value. Returns the (new or existing) node to which the attribute was added.

dump()

enter(*path*)
Enters the given node. Creates it if it does not exist. Returns the node.

leave()
Returns to the node that was selected before the last call to enter(). The history is a stack, to the method may be called multiple times.

open(*path*)
Creates and enters the given node, regardless of whether it already exists. Returns the new node.

serialize()

Gelatin.generator.Json module

class Gelatin.generator.Json.**Json**
Bases: object

serialize_doc(*node*)

Gelatin.generator.Xml module

class Gelatin.generator.Xml.**Xml**
Bases: object

serialize_doc(*node*)

serialize_node(*node*)

Gelatin.generator.Yaml module

class Gelatin.generator.Yaml.**Yaml**
Bases: object

serialize_doc(*node*)

Gelatin.generator.Yaml.**represent_ordereddict**(*dumper*, *data*)

Python Module Index

g

Index

Symbols

A

B

C

D

E

G

J

L

N

O

P

R

S

T

X

Y