Reslang System Design

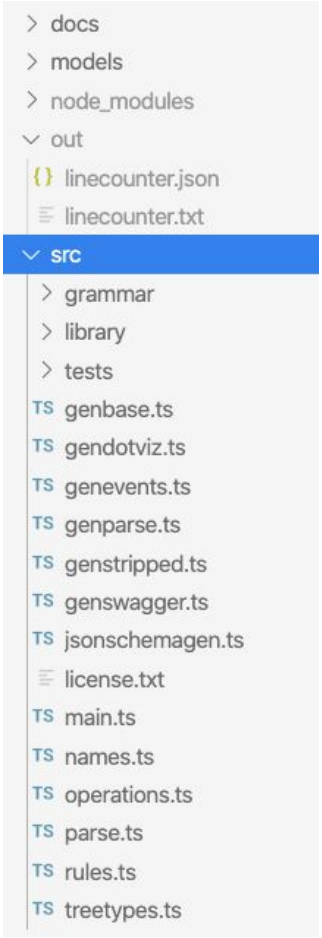
Google docs link

Reslang is a typical transpiler - converting from one language (the Reslang DSL) into a set of others (OpenAPI, AsyncAPI, JSON schema, diagrams). It is organized into the following stages:

1. Parsing: parsing reslang files into an abstract syntax tree (AST)
2. Analyzing: applying rules and heuristics to determine if the API spec is well formed
3. Transforming: generating intermediate structures
4. Translating: writing out as another specification (e.g. OpenAPI)

File Organization

Reslang itself is pretty simple, and fairly small - around 4k LOC of typescript. The picture below shows the top level organization of files:



The directories are:

/docs - contains the README.md and all other docs + pics

/models - contains a set of directories of reslang examples, along with snapshots of OpenAPI, AsyncAPI, diagram, parser output. This allows us to compare the output of many scenarios after iterating on the software. The models and snapshots play a crucial role in testing. They cover a dizzying array of scenarios, so be careful when removing something like, say, an array restriction from 0..100 - it may be the only place it is tested.

/src - the source files all live here. Main.ts is obviously the entry point, and interprets the cmd line etc. Parse.ts contains the code to drive the parser. Genbase.ts contains the base class called BaseGen - this class contains the shared analysis and transformation logic. Then we have genswagger.ts which contains the SwagGen class, which inherits from BaseGen, and translates into OpenAPI. Similarly genevents.ts contains EventsGen, which also inherits from BaseGena and translates into AsyncAPI.

Treetypes.ts contains the definitions corresponding to the abstract syntax tree - the job of parse.ts is to parse the reslang files, and creat the structures from treetypes.ts. After this point, we are dealing with typed structures.

/src/grammar - Reslang uses Pegjs, a javascript parsing library analogous to yacc/lex. The pegjs files live in this directory. Pegjs doesn't provide a way to split out a grammar into multiple files, so parse.ts joins all the files in this directory before doing parsing.

/src/library - Reslang is configurable, and the files that allow this configuration live here. Local.reslang is automatically included in every specification, it contains the definition of StandardError - there is no ability to override this via command line switches. Servers.reslang contains the default server block specifications, these can be overridden by server blocks in your reslang spec. Rules.json contains parameters for rule checking reslang specifications & defaults - such as maximum resource depth (default 2) etc. The rules file used can be overridden by a cmd line parameter. Note that defaults like pagination approach live also in this file.

/src/tests - this contains the (mostly) unit & functional tests written using jest. Some are pure unit tests - e.g. names.test.ts checks that the name handling routines are correct. Some are tests that iterate on all the models (see above) and save snapshots of the various specs generated from the models, and compare them.

To run the tests, simply type this at the top level. There are over 150 tests: yarn test

./regenerate-test-data.sh - this file runs over the models and regenerates all the output data such as OpenAPI specs etc. The snapshots are saved in /testdata directories under each model directory - under names like swagger.expected, asyncapi.expected etc.

Testing & Release Loop

The normal process of adding a feature involves:

1. Adjusting the grammar files to incorporate the feature (/src/grammar)
2. Writing the feature and some isolated unit tests (/src, /src/tests)
   (e.g. adding array multiplicity)
3. Creating or amending a model spec which exercises this feature (/models)
   (e.g. adding an attribute like: `names: string[1..10]`)
4. Regenerating the model snapshots: ./regenerate-test-data.sh
5. Diffing the old and new snapshots
6. Running the tests: yarn test
7. When we are certain the new snapshots are correct, committing

Releasing involves bumping the version in main.ts.

Processing Stages:

Parsing

/src/parse.ts contains the parsing stage, and it is fairly straightforward. It initially joins all of the grammar files to make a single spec:

```
// grammar is split into several parts
const grammar =
   readFile(__dirname, "grammar", "main.pegjs") +
   readFile(__dirname, "grammar", "tags.pegjs") +
   readFile(__dirname, "grammar", "servers.pegjs") +
   readFile(__dirname, "grammar", "rest.pegjs") +
   readFile(__dirname, "grammar", "events.pegjs") +
   readFile(__dirname, "grammar", "diagrams.pegjs") +
   readFile(__dirname, "grammar", "attributes.pegjs") +
   readFile(__dirname, "grammar", "base.pegjs")
```

It then parses each file in the directory passed on the cmd line, and turned it into a parse tree:

```
export interface IParseTree {
   namespace: any[]
   imports: any[]
   servers: any[]
   tags: any[]
   definitions: any[]
   diagrams: any[]
```

```
            docs: any[]
        }
```

At this point, all the definitions are basically json structures, returned from the pegjs grammar. E.g. a grammar rule like:

```
event = _ comment:description? _ "event" _ name:resname _ "{" _
        header:header? _ payload:payload? _
    "}" _ ";"? _ {

        return {
            category: "definition",
            kind: "event",
            type: "event",
            comment: comment,
            parents: [],
            short: name,
            header: header,
            payload: payload}
    }
```

Will return a json structure shown in the return statement when it matches with an “event” definition in the reslang spec. This will land up in the “definitions” field of the parse tree.

BaseGen will then use the parse tree and “cast” the json structures into compatible typescript structures from treetypes.ts:

```
protected namespace!: INamespace
protected servers!: IServers
protected defs: AnyKind[] = []
protected tags: ITag[] = []
protected diagrams: IDiagram[] = []
```

This can only happen because the json structures returned in the grammar are identical to the interface definitions in treetypes.ts.

Pegjs and the Reslang Grammar

Pegjs is a standard parser generator - covered well in the project documentation and intros. There is also an interactive playground for experimentation. There isn’t really much more to note about the PEG approach, other than its main oddity - it doesn’t differentiate between whitespace and other definitions, unlike yacc. As such, we defined base whitespace skipper rules as _ and __ in base.pegjs:

```
        // whitespace or comment
        _  = ([ \t\r\n]+ / comment)*
```

```
// mandatory separation
__ = ([ \t\r\n]+ / comment)+
```

You will see underscores placed liberally in other definitions to allow for whitespace between elements.

The toplevel definition in main.pegjs covers nearly all the allowed constructs:

```
reslang = (namespacedefinition / import / servers / tag / resource /
           subresource / action / structure / enum / event /
           produces / consumes / diagram / docs)*
```

Let's look at the definition of a resource, inside rest.pegjs:

```
resource = _ comment:description? _ future:"future"? _
       singleton:"singleton"? _
       type:("configuration-resource" / "asset-resource" / "resource" / "request-resource") _
       respath:noparentrespath _ "{" _ attributes:attributes? _ operations:operations? _
       events:events? _ "}" _ ";"? _ {

   return {
       category: "definition",
       kind: "resource-like",
       comment: comment, future: !!future, singleton: !!singleton, type: type,
       attributes: attributes, operations: operations, events,
       parents: [], short: respath.short}
}
```

This is fairly straight forward - you can have an optional comment, followed by an optional future keyword, an optional singleton, and then a resource type (resource, request-resource etc), followed by a name (respath) a "{" then attributes, operations, events, and "}" and optional ";".

This will return the json structure shown, which corresponds to the IResourceLike interface from treetypes.ts:

```
export interface IResourceLike extends IDefinition {
   kind: "resource-like"
   namespace?: string
   attributes?: IAttribute[]
   operations?: IOperation[]
   events?: IEventOperation[]
   singleton?: boolean
   future?: boolean
   async?: boolean
   bulk?: boolean
   … }
```

Analyzing & Transforming

Genbase.ts contains most of the analysis and transforming code, encapsulated by this set of calls:

```
this.processDefinitions()
this.checkMandatoryRules()
this.checkOperationOptions()
this.checkConfigurableRules()
if (expandInlines) {
    this.expandInlines()
}
```

processDefinitions() reads in each reslang file in the directory (and sub-directories), calls into the parser to get the parse tree, and stores the definitions in member variables. It also handles any importing of other reslang modules, but loading those definitions into the tree also, but marking them as secondary. It also loads the local.reslang library file.

Expanding inlines basically takes any inline variables and propagates the fields of their structures up one level.

BaseGen also understands types, and what is a primitive versus a complex type.

Translating

Translation happens using the following classes:

SwagGen (genswagger.ts)	Translates AST into OpenAPI
EventsGen (genevents.ts)	Translates AST into AsyncAPI
ParseGen (genparse.ts)	Translates AST into a printable parse tree
JsonSchemaGen (genjsonschema.ts)	Translates AST into JSON schema
StripGen (genstripped.ts)	Translates AST into pretty-printed HTML without comments or documentation, for ease of review
DotvizGen (gendotviz.ts)	Translates AST into a resource diagram

Let’s look at the Swagger generator in more detail. SwagGen obviously inherits most of its functionality from BaseGen. It sets up a javascript object called “swag” which is defined as:

```
const tags: any[] = []
const paths: any = {}
const schemas: any = {}
const parameters: any = {}
const servers: Array<any> = []
const swag: object = {
    openapi: "3.0.1",
    info: {
        title: this.namespace.title,
        description: this.translateDoc(this.namespace.comment),
        version: this.namespace.version
    },
    servers,
    tags,
    paths,
    components: {
        parameters,
        schemas
    }
}
```

Clearly this can easily be output directly as json, or translated into yaml, and it will be a valid OpenAPI file. The job then is to fill in the different parts.

The rest is fairly straightforward with the exception of the follow logic. This logic, encapsulated in the follow() function, basically ensures that only the definitions used in the API are actually included. You can, for instance, add in any number of structures, but only the ones that are actually used by resources are included in the OpenAPI spec. This logic is recursive - it starts from the toplevel resources and subresources first, and goes down marking the .generateXXX fields of each definition.

.generateInput means that the structure will be included for a POST, and so on. This logic is a bit tricky and needs careful attention before using it. In a way it is optional - it's purpose is to minimize the size of the file, which can easily lead to errors of omission. If every definition were included, the OpenAPI spec would be correct, but would have extra, unused elements potentially.

Testing & Models

The 24 main models are listed below:

```
edmissing multi gendiagram servers linked databuyer eventing dataset
checkrules privacy optionality authorization complex-resource
patchable direct2dist distribution file request simple-resource
singleton stringmaps upversion multiplicity namespaces
```

Each one has a set of scenarios encoded within it, and the scenarios are usually described in the name. E.g. multiplicity has a resource which defines a set of attribute arrays with different max and min bounds.

Regenerate-test-data.sh is a small shell script which regenerates all the snapshot data living in the models/X/testdata directories.