oto

Go driven rpc code generation tool for right now.

- 100% Go
- Describe services with Go interfaces (with structs, methods, comments, etc.)
- Generate server and client code
- Production ready templates (or copy and modify)

Who’s using Oto?

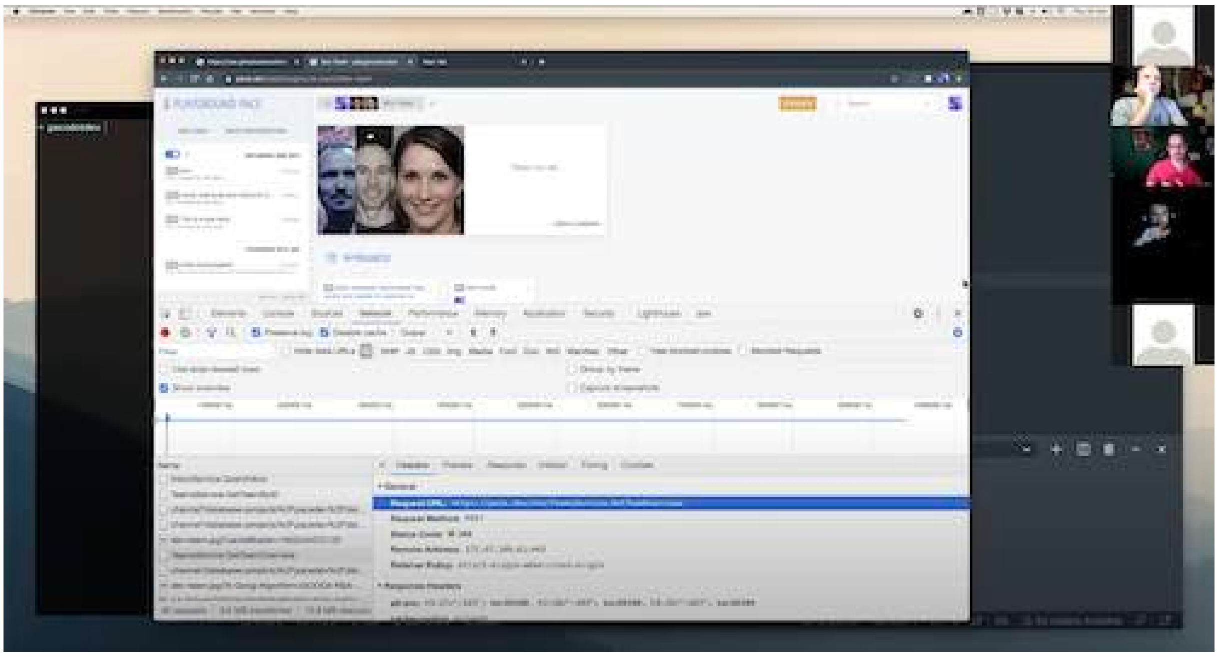
- Grafana Labs, IRM tool
- Pace.dev
- Firesearch.dev

Templates

These templates are already being used in production.

- There are some official Oto templates
- The Pace CLI tool is generated from an open-source CLI template

Learn



- VIDEO: Mat Ryer gives an overview of Oto at the Belfast Gophers meetup
- BLOG: How code generation wrote our API and CLI

Tutorial

Install the project:

```
go install github.com/pacedotdev/oto@latest
```

Create a project folder, and write your service definition as a Go interface:

```go
// definitions/definitons.go
package definitions

// GreeterService makes nice greetings.
type GreeterService interface {
	// Greet makes a greeting.
	Greet(GreetRequest) GreetResponse
}

// GreetRequest is the request object for GreeterService.Greet.
type GreetRequest struct {
	// Name is the person to greet.
	// example: "Mat Ryer"
	Name string
}

// GreetResponse is the response object containing a
// person's greeting.
type GreetResponse struct {
    // Greeting is the greeting that was generated.
    // example: "Hello Mat Ryer"
    Greeting string
}
```

Download templates from otohttp

```
mkdir templates \
    && wget https://raw.githubusercontent.com/pacedotdev/oto/master/otohttp/templates/server.go.plush -q -O ./templates/server.go.plush \
    && wget https://raw.githubusercontent.com/pacedotdev/oto/master/otohttp/templates/client.js.plush -q -O ./templates/client.js.plush
```

Use the `oto` tool to generate a client and server:

```
mkdir generated
oto -template ./templates/server.go.plush \
    -out ./generated/oto.gen.go \
    -ignore Ignorer \
    -pkg generated \
    ./definitions
gofmt -w ./generated/oto.gen.go ./generated/oto.gen.go
oto -template ./templates/client.js.plush \
    -out ./generated/oto.gen.js \
    -ignore Ignorer \
    ./definitions
```

- Run `oto -help` for more information about these flags

Implement the service in Go:

```
// greeter_service.go
package main

// GreeterService makes nice greetings.
type GreeterService struct{}

// Greet makes a greeting.
func (GreeterService) Greet(ctx context.Context, r GreetRequest) (*GreetResponse, error) {
    resp := &GreetResponse{
        Greeting: "Hello " + r.Name,
    }
    return resp, nil
}
```

Use the generated Go code to write a `main.go` that exposes the server:

```go
// main.go
package main

func main() {
	g := GreeterService{}
	server := otohttp.NewServer()
	server.Basepath = "/oto/"
	generated.RegisterGreeterService(server, g)
	http.Handle(server.Basepath, server)
	log.Fatal(http.ListenAndServe(":8080", nil))
}
```

- The `otohttp.Server` performs its own routing and so has a `Basepath` field which you should use when you route the handler.

Use the generated client to access the service in JavaScript:

```javascript
import { GreeterService } from "oto.gen.js";

const greeterService = new GreeterService();

greeterService
  .greet({
    name: "Mat",
  })
  .then((response) => alert(response.greeting))
  .catch((e) => alert(e));
```

Use `json` tags to control the front-end facing name

You can control the name of the field in JSON and in front-end code using `json` tags:

```go
// Thing does something.
type Thing struct {
	SomeField string `json:"some_field"
}
```

- The `SomeField` field will appear as `some_field` in json and front-end code
- The name must be a valid JavaScript field name

Specifying additional template data

You can provide strings to your templates via the `-params` flag:

```
oto \
    -template ./templates/server.go.plush \
    -out ./oto.gen.go \
    -params "key1:value1,key2:value2" \
    ./path/to/definition
```

Within your templates, you may access these strings with `<%= params["key1"] %>`.

Comment metadata

It's possible to include additional metadata for services, methods, objects, and fields in the comments.

```
// Thing does something.
// field: "value"
type Thing struct {
    //...
}
```

The `Metadata["field"]` value will be the string `value`.

- The value must be valid JSON (for strings, use quotes)

Examples are officially supported, but all data is available via the `Metadata` map fields.

Examples

To provide an example value for a field, you may use the `example:` prefix line in a comment.

```
// GreetRequest is the request object for GreeterService.Greet.
type GreetRequest struct {
    // Name is the person to greet.
    // example: "Mat Ryer"
    Name string
}
```

- The example must be valid JSON

The example is extracted and made available via the `Field.Example` field.

Open API

To work on the Open API spec, you might find this command helpful:

```
oto -template ./otohttp/templates/openapi.yaml.plush -out openapi.yaml
    -v -ignore Ignorer ./parser/testdata/services/pleasantries
```

Contributions

Special thank you to:

- @mgutz - for struct tag support
- @sethcenterbar - for comment metadata support