

Guião para trabalhar com ITOI

1 - Visão Geral

Requisitos:

- Conseguir fazer e perceber o [tutorial](#) do Theia
- Conseguir fazer e perceber o [tutorial](#) do Langium
- Noções básicas de Docker.

ITOI é uma aplicação containerizada, por isso toda a parte do processo de dar BUILD do projeto está descrito no Dockerfile.

A pasta ide é onde está o código para o IDE e as pastas plugin é onde estão os language server para as 2 linguagens ASL e RSL e outros plugins utilizados.

Nome	Data de modificação	tipo	tamanho
.git	07/11/2023 17:16	Pasta de ficheiros	
configs	15/10/2023 19:47	Pasta de ficheiros	
db	15/10/2023 20:02	Pasta de ficheiros	
ide	15/10/2023 19:47	Pasta de ficheiros	
pack	15/10/2023 19:47	Pasta de ficheiros	
plugins	15/10/2023 19:47	Pasta de ficheiros	
.dockerignore	15/10/2023 19:47	Ficheiro DOCKERI...	1 KB
.gitignore	15/10/2023 19:47	Arquivo Fonte Git ...	1 KB
app.json	15/10/2023 19:47	Arquivo Fonte JSON	1 KB
createDatabase.sql	15/10/2023 19:47	Arquivo Fonte SQL	3 KB
docker-compose.yml	15/10/2023 19:47	Arquivo Fonte Yaml	1 KB
Dockerfile	15/10/2023 19:47	Ficheiro	4 KB
heroku.yml	15/10/2023 19:47	Arquivo Fonte Yaml	1 KB
license	15/10/2023 19:47	Ficheiro	2 KB
package.json	15/10/2023 19:47	Arquivo Fonte JSON	1 KB
readme.md	15/10/2023 19:47	Arquivo Fonte Ma...	1 KB
runlocal.cmd	15/10/2023 19:47	Script de Comand...	1 KB
startup.sh	15/10/2023 19:47	Arquivo Fonte SH	1 KB

Commandos relevantes:

- docker build . -t itoi
- docker compose up -p 3000:3000 api sh
- yarn theia start --plugins=local-dir:../plugins --hostname 0.0.0.0 --port 3000 --no-cluster

Para correr, é necessário correr os comandos nesta ordem. O Docker build passa a tag "itoi" porque depois é a referência utilizada no docker-compose. Após o docker compose up vais ter acesso ao container na pasta "browser-app" e é nesta pasta que corres o yarn theia start.

2 - IDE

O código do IDE encontra-se dentro da pasta `ide\itlingo-itoi\src` e está dividido em 3 pastas diferentes:

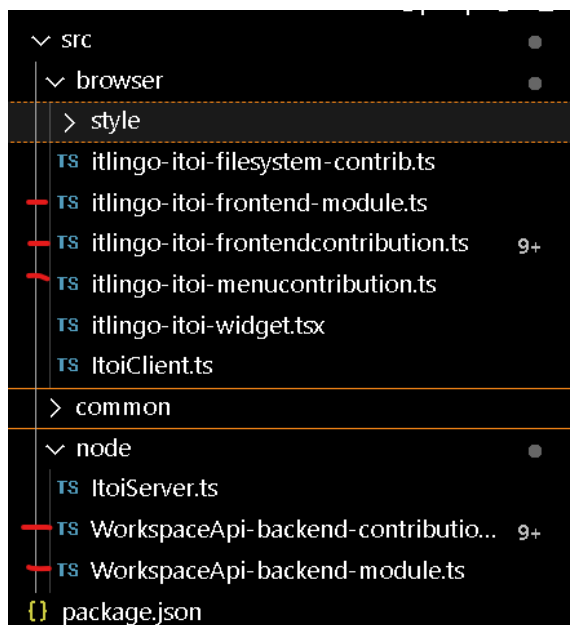
- browser : código que corre no browser do cliente, é aqui que podes modificar acções dos botões ou adicionar botões
- node : código que corre no servidor, é aqui que adicionas endpoints por exemplo.
- common : constantes que queiras partilhar entre as duas pastas.

A framework Theia por defeito tem 2 target builds, um para browser e outro para electron (que não está a ser utilizado). Por isso temos a pasta browser-app e electron-app.

A pasta plugin é para onde são copiados os plugins após serem construídos (ver Dockerfile).

Theia utiliza dependency injection, o ficheiro `browser\itlingo-itoi-frontend-module.ts` é onde se faz o bind entre as classes do Theia com as nossas classes. A classe que faz bind com **FrontendApplicationContribution** é a principal para trabalhar com interação com utilizador e **CommandContribution** para alterar comportamentos dos botões ou do lado direito do rato.

Para o backend temos o WorkspaceApi-backend-contribution para ver onde está a ser feito os endpoints, conexão com a base de dados e o filewatcher que atualiza os ficheiros para a base de dados.

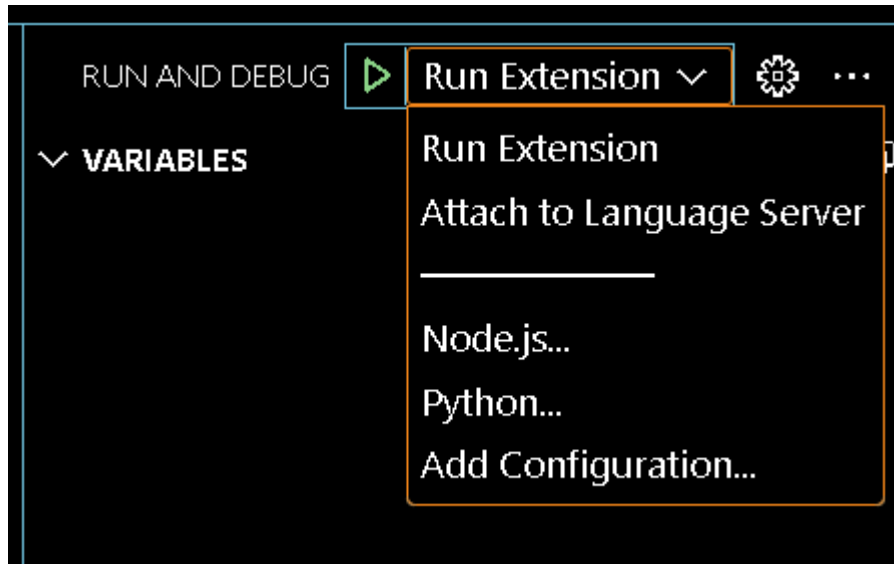


O ficheiro **ItoiServer.ts** é onde já se tem um serviço de websockets a funcionar, por isso de preferência se for para receber informação do cliãente, ver se é possível por aqui.

3 - Plugins

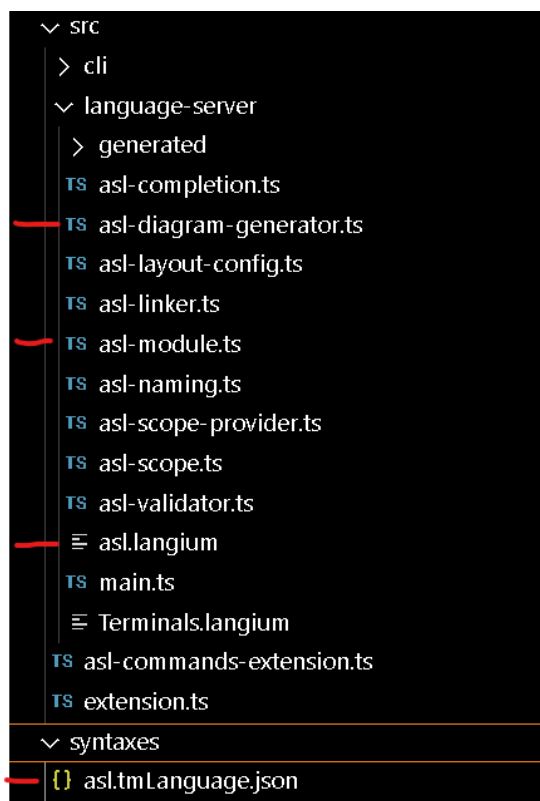
Para o plugin de RSL e ASL a explicação é a mesma, por isso vou usar o ASL como exemplo.

Desenvolvido com a framework Langium, é um projecto de uma extensão de vscode, que dá para testar directamente no vscode.



Tem que se fazer Run Extension e depois Attach to Language Server, para se conseguir utilizar os breakpoints.

Como Theia, utiliza dependency injection, por isso para alterar algo, é preciso ver qual a class que queremos trabalhar. asl.langium é o ficheiro de gramática que foi convertido do xtext.



Comandos relevantes:

- yarn (para buscar as bibliotecas);
- yarn run langium:generate (para gerar os ficheiros na pasta language-server\generated, só é necessário se alterar mos o ficheiro asl.langium)

Nota que a correr langium:generate, copiar para fora o asl.tmLanguage.json, para não ser substituido, porque foram feitas alterações e a geração substitui o que lá estava antes. (ou alterar no package.json o script langium:generate para copiar para fora o ficheiro e quando acaba o generate copiar de novo lá para dentro).

Caso seja necessário fazer de novo a conversão de xtext para langium [link](#).

4- Diagram Generation

Para a geração de diagramas foi utilizado o Sprotty.

Em cada uma dos plugins já existe a class para trabalhar com a lógica do diagrama de `rsl-diagram-generator.ts` ou `asl-diagram-generator.ts`.

Na root temos a pasta pack com o ficheiro `webview.js`, este já é um ficheiro construído do seguinte projecto de exemplo:

<https://github.com/eclipse-sprotty/sprotty-vscode/tree/master/examples/states-webview>

Não chegou a ser feito, mas se for para adicionar elementos aos diagramas, deve-se ter que alterar esta webview, que este é o ficheiro aberto no browser para o cliente e mostra os elementos.

```
TS rsl-diagram-generator.ts 2 X
plugins > rsl-vscode-extension > src > language-server > TS rsl-diagram-generator.ts > ...
14  * SPDX-License-Identifier: EPL-2.0 OR GPL-2.0 WITH Classpath-exception-2.0
15  *****/
16
17  import { GeneratorContext, LangiumDiagramGenerator } from 'langium-sprotty';
18  import { SEdge, SLabel, SModelRoot, SNode } from 'sprotty-protocol';
19  import { Actor, Model, UseCase, View } from './generated/ast';
20
21  export class RslDiagramGenerator extends LangiumDiagramGenerator {
22
23      protected generateRoot(args: GeneratorContext<Model>): SModelRoot {
24          const { document } = args;
25          const root = document.parseResult.value;
26          let views = this.fetchListOfViews(root, args);
27          const { sNode, argsNodes } = this.mapViewsToSNodes(views, args);
28          let sEdges = this.fetchListOfEdges(views, argsNodes);
29
30          return {
31              type: 'graph',
32              cssClasses: ['sprottyCss'],
33              id: 'root',
34              children: [...sNode, ...sEdges]
35          };
36      }
37
38      mapViewsToSNodes(views: View[], args: GeneratorContext<Model>):
39      {sNode: SNode[],
40      argsNodes: GeneratorContext<Model>} {
41          let nodes: SNode[] = []
42          for(const v of views){
```