Estrutura de software – E3G::SM

* Engine: Node.js
* Framework: Electron
* Linguagem: Javascript
* Banco de dados: SQLite 3
* Estrutura de backend: MVC
* Framework frontend: AngularJS 1.6

Funcionamento do Node.js e Electron

O **Electron é um módulo JS** com diversas funções e propriedades que se utilizam do Node.js para criação de janelas e manipulação de elementos gráficos do sistema operacional, para que isso seja possível, o Electron é executado sobre o Node.js.

O Node.js utiliza o mecanismo **JavaScript V8 do Chrome** para executar fontes JS. Ele disponibiliza ao programador módulos que podem ser utilizados para interagir com o SO e por sua linguagem principal ser JavaScript a sua estrutura é **orientado a eventos**.

Orientação a eventos/objeto

O JavaScript é uma linguagem **não tipada orientado a eventos**, ou seja, ele apresenta um fluxo de execução parte linear e outra paralela.

Existe dois tipos de execução de funções, as funções Síncronas e as Assíncronas, a síncronas são aquelas que o programador as invoca e aguarda o retorno para que este seja armazenado em uma variável ou seja utilizado numa condição, quando o programa executa estas funções o fluxo deste código é bloqueado e só continuará quando a função retornar, este é efeito é chamado de **eventos bloqueantes**. As funções assíncronas são executas pelo código principal e como parâmetro é exigido uma função como **call-back**, quando a função finalizar esta função passada será executada, desta forma, a execução desta tarefa ocorrerá paralelamente ao fluxo principal. Exemplo:

let idade = null;

idade = getIdadeSync(1998 /\*ano nasc\*/); // função síncrona

console.log("Idade: ${idade}"); // "Idade: 19"

let idade2 = null;

getNomeAsync(1998, function (return) { // função assíncrona

console.log("Idade: ${idade}"); // "Idade: 19"

});

console.log("Idade: ${idade}"); // "Idade: " null

O JavaScript tem suporte à objetos, prototipada, onde um array é considerado um objeto de índices numéricos e um array associativo (aquele que o programador associa uma chave tipo *String* para determinado valor) também é considerado um objeto, uma função também poderá ser considerada como objeto se houver propriedade internas inseridas. Exemplos:

// Objeto Array Simples

let obj1 = ["Zero","Um","Dois"];

console.log(typeof obj1); // Out: "object"

// Objeto Array Associativo

let obj2 = {

"a":"A",

"b":"B",

"1": {

"UM"

}

};

console.log(typeof obj2); // Out: "object"

// Objeto Função - prototype

function func1 (param) {

console.log(param);

reutn this; // this => func1

this.func2 = () => {

console.log("Sou uma função");

};

this.obj1 = {"idx1":"Eu sou uma propriedade do objeto"}

}

console.log(typeof func1); // Out: "function"

func1("sou um parametro"); // Out: "sou um parametro"

func1.func2(); // Out: "Sou uma função"

func1("oi").func2(); // Out: "oi", "Sou uma função"

console.log(typeof func1.obj1.idx1); // Out: "string"

let newFunc = new func1;

console.log(newFunc);

/\* Out:

newFunc: {

obj1: {...},

func2: function

}

\*/

Responsabilidade

A estrutura MVC tem como principal objetivo manter a clara distinção de responsabilidade, ela é comumente utilizada em programação Web e existem diversos frameworks que facilitam a sua utilização, como o Laravel, CakePHP, Codeigniter, Phalcon, dentro muitos outros. O Node.js apresenta o Express como um dos mais famosos frameworks para aplicações Web. No entanto, por estarmos utilizando o Electron que tem como foco produção de aplicativos híbridos para desktop e por ser relativamente recente, não existem frameworks MVC disponíveis. Para cada projeto o programador deve determinar qual a melhor estratégia estrutural, há diversas possibilidades. Em nossa estrutura decidimos criar um Core que terá como objetivo centralizar tarefas e a comunicação entre a Models, Views e Controllers por meio de seu objeto.

Para que o sistema funcione corretamente é necessário que o programador respeite a responsabilidade e nomenclatura dos arquivos do programa.

As **Models** têm como único objetivo realizar as transações no banco de dados, para qualquer solicitação de informações do banco de dados quem irá executa-la será a Model responsável para determinada tabela. Cada tabela terá a sua Model, onde esta deverá conhecer estritamente a sua estrutura, não é permitido que ela realize *insert, update* ou *delete* em outras tabelas, em relação a *selects* a Model apenas poderá acessar outras tabelas se estas fazerem parte de ***joins***, se comporem o ***from*** essa transação não é aceita.

As **Controllers** são as responsáveis por realizarem as validações de regras de negócio e atenderem qualquer solicitação externa, ou seja, se uma **View** necessitar de qualquer informação do banco de dados, esta deverá solicitar à sua respectiva Controllere por sua vez ela validará a solicitação e se comunicará com a **Model** da tabela necessária, realizada a *query*o retorno será levado para a Controller e da Controller para a View.

As **Views** são a UI, compostas por HTML, CSS e JS. A montagem da página é feita pelo módulo ***EJS*** (Embedded JS – JS Embutido), para que este módulo seja utilizado os arquivos de layout não podem ser ***.html***, mas sim ***.ejs***. Ele não é responsável por realizar validação de regras de negócio, a sua função é interpretar o retorno da Controller e montar a página.

Como foi dito, o Frontend será composto pelo AngularJS, a sua estrutura arquitetônica não é bem definida, depende muito de como o programador a utilizará (MVW – Model – View - Whatever), mas em questões de boas práticas a base é o MVVM (Model – View - ViewModel). O AngularJS contém o seu próprio núcleo de arquivos, ele será responsável por manipular os elementos DOM/UI, realizar pré-validações dos formulários antes de enviar as informações para o Core central (mais especificamente para as Controllers) e controlar as rotas internas das páginas por meio de URLs, para cada URL o AngularJS realizará uma solicitação ao Core. Essa estratégia de roteamento da UI decorre do seu ***template***, para fins de elegância é carregado uma ***partialView*** e não a página inteira e para que isso seja possível o AngularJS deve administrar as solicitações.

Diretórios

./package.json

./app.json

./config.json

./main.js

./node\_modules

./app

./app/controller

./app/model

./app/model/database/database.db

./app/wiondow

./app/core

./app/helper/ ./app/viewCore/...

./app/assets/

***package.json***

Este arquivo contém todas as informações do aplicativo, todos módulos e dependências do Node.js.

***main.js***

O arquivo ***main.js*** é o primeiro a ser executado, segundo especificação em ***package.json***. A sua responsabilidade é instanciar os módulos do Node.js e do ***Electron*** necessários para inicializar o ***Core***; montar um array com a lista de todos os diretórios do programa com base no ***\_\_dirname***; o main.js é o único que tem a função de finalizar o processo principal do programa; iniciar o Core passando todas as dependências.

***app.json / config.json***

Contém as configurações gerais do programa, lista de diretórios, Controllers e Models que devem ser carregados, diretório do banco de dados dentre outras informações.

***controller***

Neste diretório deverá estar todas as Controllers do backend e respeitará a seguinte nomenclatura:

./app/controller/[nomeDaContrller]/[nomeDaController\_Maiúscula]Controller.js

Exemplo:

./app/controller/**main**/**MainController**.js

Nota-se que a pasta da Controller deverá ser tudo minúsculo e o script principal dela deverá conter a primeira letra maiúscula concatenado com "Controller.js"

***model***

Neste diretório deverá estar todas as Models do backend e respeitará a seguinte nomenclatura:

./app/model/[nomeDaTabela]/[nomeDaTabela\_Maiúscula]Model.js

Exemplo:

./app/model/**config**/**ConfigModel**.js

A regra é parecida com a da Controller, além do fato de que o nome da model deverá ser o mesmo nome da tabela que é responsável.

***helper***

Os helpers são módulos que tem como objetivo auxiliar em qualquer parte do código em tarefas como validação de e-mail/IP/CPF/CNPJ dentre outras funções, como buscar o logradouro a partir do CEP, etc.

A estrutura da pasta será:

./app/helper/[nomeDoMódulo]/index.js

Nota-se que o nome da pasta do helper poderá ter carácteres maiúsculos e minúsculos, a única restrição são espaços. O Arquivo principal do módulo deve ser obrigatoriamente o index.js.

***viewCore***

Esta pasta contém todos os arquivos do Core do template – AngularJS. Todos serão utilizados exclusivamente pelas Views.

***assets***

Os assets são todas as imagens, fonts, css e js adicionais utilizados pelo programa.

***window***

*Nesta pasta estará localizado o* **index.ejs**e ***partialViews*** das janelas, a sua estrutura de pastas é:

./app/window/[nomeDaJanela]/index.ejs

./app/window/[nomeDaJanela]/commom/[...].ejs

./app/window/[nomeDaJanela]/content/[...].ejs

Em ***common*** está o template comum em todas as páginas, como barra de navegação, rodapé e etc.

Em ***contente*** está todas as ***partialViews*** das páginas.

Recomenda-se que o **nome da janela** não contenha espaços e carácteres especiais.

***core***

No core contém o arquivo ***linkRenderCore.js*** responsável por ligar a View com o Core e o arquivo ***Core.js*** que é o Core em si. As suas funções são:

* Core.init

**Parâmetros**

**- Electron.app**

**- Electron.BrowserWindow**

**- Electron.dialog**

**- fs (fileSystem)**

**- \_.LoDashStatic**

**- electron-router**

**- dirsList::object**

Iniciar o Core instanciando módulos necessários, validando lista de configuração e carregando controllers e models.

* Core.loader

**Parâmetros**

**- type :: tipo do módulo [node|controller|model|helper|core]**

**- \_name :: nome do módulo**

**- nameInstance :: caminho do Core::objeto para salvar o módulo**

**- toRenderCore :: caso seja especificado *nameInstance* haverá a opção de salvar também no objeto do Core da View, Core parcial**

Esta função instancia o módulo especificado conforme o tipo dele, tendo a opção salvar este módulo dentro do núcleo tornando-o global ou apenas retorna-lo.