Sails.js

Fernando Anselmo
http://fernandoanselmo.orgfree.com/wordpress/

Versão 1.0 em 6 de julho de 2020

Resumo

Sails.js [1] é um framework Web que facilita a criação de aplicativos Node.js [2] customizados a nível empresarial. Nesta apostila vamos do ZERO criar o início para um Dashboard no qual podemos controlar nossa conta bancária. Com o Sails.js é necessário conhecimento da Linguagem JavaScript a nível de outros frameworks conhecidos no mercado como o Angular.js ou o Vue.js, e todo o conhecimento pode ser aproveitado para criar as camadas de um MVC completo, além da possibilidade de usar diversos geradores automáticos e acesso aos bancos mais conhecidos do mercado.

1 Parte inicial

Sails.js (doravante chamaremos apenas de **Sails**) é um framework abrangente que segue o padrão MVC para **Node.js** projetado especificamente para permitir um rápido desenvolvimento de aplicativos do lado do servidor e a disponibilização de serviços em JavaScript. Possui uma forte arquitetura Orientada a Serviços que fornece diferentes tipos de componentes no qual utilizamos para organizar o código e separar as responsabilidades.

Figura 1: *Logo do Sails.js*

Vejamos algumas características do produto:

- É 100% JavaScript.
- É possível usar qualquer sistema de banco de dados: possui um ORM nativo, **Waterline** [5], que fornece uma camada de acesso a dados simples que funciona, não importa o banco de dados que se está usando.

- Auto-gerador de APIs REST: vem com blueprint (projetos que apresenta a melhor forma de realizar algo) que ajudam a iniciar o *backend* da sua aplicação sem escrever qualquer código.
- Suporte fácil ao WebSocket: traduz mensagens de socket recebidas.
- Políticas de segurança reutilizáveis: fornece uma segurança básica e controle de acesso baseado em funções por padrão que podem ser incrementadas com o uso de JWT.
- Pipeline de ativos flexíveis: com o **Grunt** e o **Vue.js** - significa que todo o fluxo de trabalho e recursos do *frontend* tornam-se completamente personalizáveis.

Sails fornece um benefício adicional de ser capaz de compartilhar entre o servidor e cliente. Isso pode ser muito útil, por exemplo, para implementar regras de validação de dados.

1.1 O que é o padrão MVC?

O Padrão de Projeto (*Design Pattern*) MVC define a separação de um sistema em três camadas, conforme a seguinte figura:

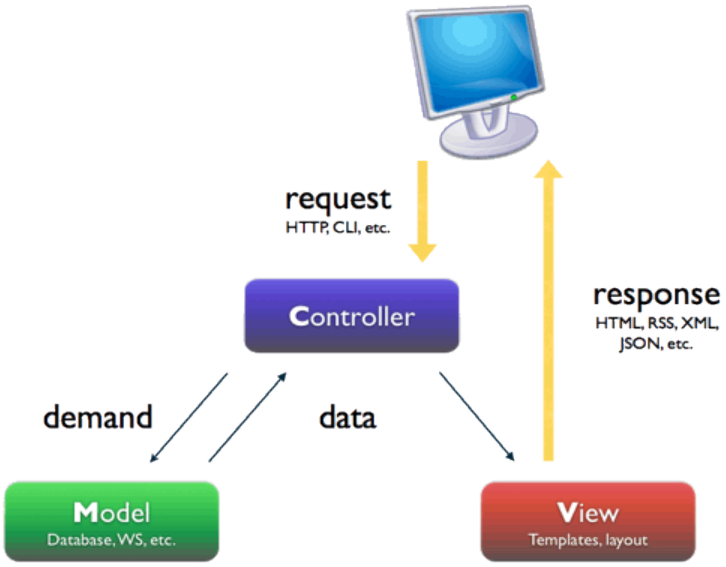

Figura 2: *Padrão de Projeto: MVC*

- **Camada de Modelo** (Model) responsável pela interação com o banco de dados.
- **Camada de Visão** (View) responsável por mostrar os dados na tela.
- **Camada de Controle** (Controller) responsável por gerenciar toda a comunicação entre Modelo e Visão. Não é permitido o relacionamento inter-camadas.

Simplificadamente pense em uma empresa na qual existe um cliente que deseja falar com um determinado funcionário, porém para acessar sua sala deve passar por uma Secretária. Esse funcionário deseja obter uma determinada informação de um arquivo, para isso deve solicitá-la a Secretária. Ou seja, ela é a controladora de tudo o que será permitido visualizar ou obter de informação da empresa.

1.2 Serviços Web

Fornecem uma série de consideráveis benefícios para o desenvolvimento de aplicativos, permitindo uma agilidade requerida pelas empresas frente às mudanças que podem ocorrer no ambiente de negócios. A maior vantagem é que provém a capacidade de permitir uma rápida construção de várias **Visões** nas mais diversas plataformas existentes.

Figura 3: *Serviços podem estar na nuvem ou não*

Por exemplo, uma vez construída a camada de serviços, podemos utilizar o **Flutter** para construir uma camada que será acessível para plataforma Mobile, uma outra em **Java/Swing** via *desktop* e uma terceira com **Angular.js** acessível pelo navegador, porém todas usam a mesma camada de Serviço.

Assim vemos como Serviços Web são utilizados para integração entre aplicações. REST (acrônimo de *Representational State Transfer*) é uma das formas para criar um Serviço Web, que utilizada com o protocolo HTTP. O conjunto de operações, suportadas pelo serviço, podem ser de quatro tipos:

- **GET**. Listar todos os registros de uma coleção ou recuperar um único identificado como 1234.
- **POST**. Adicionar um novo registro na coleção ou obter todos registros com base em um filtro.
- **PUT**. Atualizar (substituir) determinados campos de um registro identificado como 1234.
- **DELETE**. Eliminar uma coleção ou um determinado registro identificado como 1234.

REST tem se consolidado como uma base para disponibilização de negócios eletrônicos e a construção de uma rede intra/inter organizacional de aplicações colaborativas e distribuídas, onde os Serviços Web, na forma de módulos auto-contidos, são descritos, publicados, localizados e dinamicamente invocados através de camadas com diversas visões.

2 Instalação do Sails.js

Com os conceitos do Padrão MVC e REST podemos partir para o Sails. Uma vez que temos instalado o **Node.js** e o gerenciador de pacotes **NPM** [3], abrimos uma janela de comandos para instalar o Sails (lembrando que 'sudo' só é usado se estiver em ambiente Linux):

```
$ sudo npm install -g sails
```

Como editor para os códigos, recomendo o **Visual Studio Code** que permite uma boa integração, desde que seja adicionado alguns plugins:

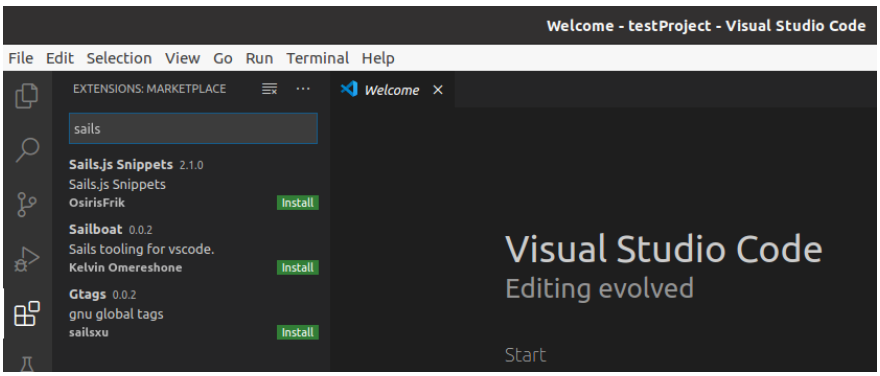

Figura 4: *Plugins do Sails no Visual Studio Code*

Os plugins "*Sails.js Snippets*"e "*Sailboat*"facilitam nosso trabalho. A grande vantagem desse editor é que além de ser leve, permite a abertura de um terminal. Ou seja, executamos todo o trabalho de editoração do código e criação de artefatos sem a necessidade de saltar de janelas.

2.1 Criar o Projeto

Para criar um projeto, abrir um terminal e colocar o seguinte comando:
`$ sails new tstSails --linker --fast`

Quando perguntado escolher opção **1. Web App**. A opção 'linker' faz com que quaisquer recursos sob a pasta /assets sejam copiados para a pasta .tmp/public pelo Grunt quando Sails for levantado. A opção 'fast' verifica se as dependências já existem, se sim não as instala novamente.

Nosso próximo passo é acessar a pasta:
`$ cd tstSails`

Instalar as dependências do Node.js:
`$ npm install`

Iniciar o sails:
`$ sails lift`

Se tudo estiver correto é possível ver o site de boas vindas no endereço: `http://localhost:1337`

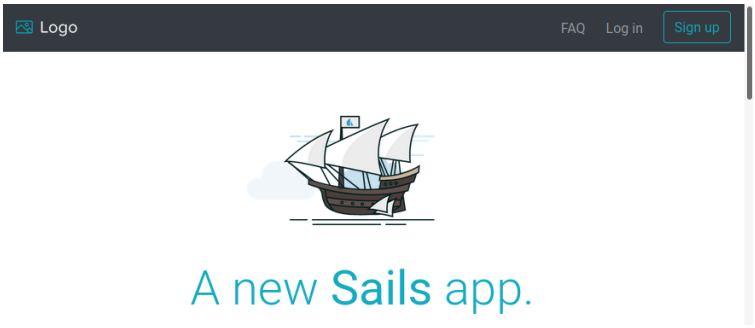

Figura 5: *Página inicial do projeto*

Para interromper o servidor, pressionar a sequencia `Ctrl+C` no terminal.

Como opção, podemos criar o projeto apenas como um provedor de Serviços Web, isto é, sem a Camada de Visão, para isso utilizamos a opção:
`$ sails new tstSails --no-frontend`

Porém como o nosso objetivo aqui é a criação de um *Dashboard* precisamos ter um projeto Web completo.

3 Geração dos Artefatos

Uma vantagem em se utilizar o Sails é a geração das camadas MVC, sem precisarmos perder tempo com configurações desnecessárias. A pasta **/api** contém a camada de *backend*. A pasta **/api/policies** encontramos as regras para o acesso do usuário da aplicação.

A pasta **/api/responses** contém arquivos como os erros do Servidor Web (404, 498, 500 entre outros). Adicionamos nessa pasta funções que lidam com tarefas específicas, como decidir como gerenciar usuários com diferentes níveis de acesso. Poderia tudo ser feito na Camada de Controle, mas não é uma boa prática termos controladores com um várias lógicas de negócios.

As camadas MVC do projeto são encontradas nas seguintes pastas:

- **Camada Modelo** que está disponível na pasta **/api/models**, arquivos padrão **.js**
- **Camada Controle** que está disponível na pasta **/api/controllers**, arquivos padrão **.js**
- **Camada Visão** que está disponível na pasta **/views**, arquivos padrão **.ejs**

Gerar a Model e a Controller:
`$ sails generate api |Nome|`

Gerar um Controlador:
`$ sails generate controller |Nome| |Ação|`

Gerar um Modelo:
`$ sails generate model |Nome| |Attribute:Type|`

Gerar uma Visão:
`$ sails generate page |Nome|`

Esse último comando gera os seguintes artefatos:

- Arquivo de visão em: `views/pages/|Nome|.ejs`
- Arquivo de controle em: `api/controllers/view-|Nome|.js`
- Arquivo para estilos CSS da página em: `assets/styles/pages/|Nome|.less`
- Arquivo para scripts JavaScript em: `assets/js/pages/|Nome|.page.js`

Devemos sempre respeitar essa forma de trabalhar (*dividir para conquistar*) e lembre-se sempre que no projeto é muito mais fácil criar do que dar manutenção, as coisas estando em seus devidos lugares torna nosso trabalho futuro menos oneroso.

3.1 Criar a Model e a Controller

Vamos por partes, parar o projeto (CTRL+C) e executar o seguinte comando:
`$ sails generate api cliente`

Dois arquivos foram criados, **/api/models/Cliente.js** e **/api/controllers/Cliente.js**. Modificar o conteúdo do primeiro arquivo para:

```
module.exports = {
  attributes: {
    nome: {
      description: 'Nome completo.',
      type: 'string',
      required: true
    },
    endereco: {
      description: 'Endereço residencial.',
      type: 'string'
    },
    idade: {
      description: 'Idade (em anos).',
      type: 'number'
    },
  },
};
```

Criamos uma modelo com 3 campos: nome e endereço com o tipo caractere e idade do tipo numérico, ao ativar o Sails esse se encarregará de informar essa mudança ao banco de dados. No caso de criar a tabela e os campos. Para expor o serviço devemos realizar alterações de segurança na pasta **/config**:

- No arquivo **blueprints.js** trocar o valor da opção *rest* para *true*
- No arquivo **policies.js** adicionar a opção *'cliente/*': true,*
- No arquivo **security.js** trocar o valor da opção *csrf* para *false*.

Essas mudanças permitem acessar os serviços REST disponíveis que foram criados (repare no entanto que o controle - segundo arquivo criado - está vazio). Ativar o Sails:
`$ sails lift`

Utilizamos um cliente REST para acessar os Serviços Web que estão expostos. Existem vários, pessoalmente prefiro o **Postman** [4]:

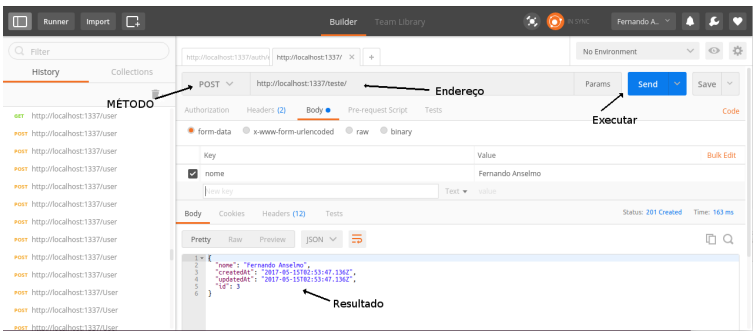

Figura 6: *Aplicativo Postman*

São os seguintes serviços disponibilizados:

- **http://localhost:1337/cliente**, método GET que traz todos os clientes cadastrados
- **http://localhost:1337/cliente/1**, método GET que traz o clientes com ID igual a 1
- **http://localhost:1337/cliente**, método POST que ao ser passado as variáveis nome, endereco e idade no BODY de um formulário será adicionado um novo registro na tabela cliente.
- **http://localhost:1337/cliente/1**, método PUT que ao ser passado as variáveis nome, endereco e idade no BODY de um formulário será modificado os dados do cliente com ID igual a 1.
- **http://localhost:1337/cliente/1**, método DELETE que exclui o registro da tabela cliente com ID igual a 1.

3.2 Arquivos de Configuração

Já modificamos alguns arquivos de configuração, existem outros importantes que são bastante utilizados:

- **/config/sockets.js**: proceder as conexões de socket.
- **/config/routes.js**: definir as URLs das Visões e *endpoints* para os métodos da Camada de Controle.
- **/config/datastores.js**: especificar os conectores do Banco de Dados, além disso organiza como migrar os dados.

Por padrão o Sails acessa um ORM (*Object Relational Mapper*) chamado **Waterline** que normalmente é utilizado para realização de testes e construção básica do sistema. Este é um excelente banco de dados, mas normalmente as empresas preferem Bancos mais conhecidos e manuseáveis por SGBD, porém não se preocupe pois é possível (através do uso de adaptadores) se conectar a, basicamente, todos os bancos conhecidos do mercado.

No arquivo **/config/models.js** é possível definirmos o modo como a base de dados trata os dados:

- **safe**. Nunca migrar automaticamente a base de dados (usado por padrão, se nada for definido).
- **alter**. Migrar os dados, mas manter os dados já existentes (isso é usado quando alteramos um modelo já existente).
- **drop**. Cada vez que o Servidor for reiniciado, eliminar TODOS os dados e reconstruir os modelos.

Outro ponto interessante nesse arquivo e a especificação dos atributos "padrões"que devem ser criados para todos os modelos, como **createAt**, **updateAt** e **id**. Assim não precisamos colocar estes atributos em cada um dos modelos.

Também podemos subir o servidor com uma dessas opções. Por exemplo:

```
$ sails lift --models.migrate='drop'
```

Essa opção é interessante para eliminar os dados e criar uma base nova, ao invés de colocá-la na configuração o que pode apresentar um grande perigo.

4 Acessar o MySQL

A parir desse ponto, utilizaremos o SGBD Relacional MySQL. O que mais me impressiona no Sails é a facilidade com a qual podemos conseguir isso. Obviamente devemos ter o MySQL, o meio mais simples é baixar a imagem oficial através do Docker:

```
$ docker pull mysql
```

Criar uma instância do banco em um contêiner:

```
$ docker run --name meu-mysql -e MYSQL_ROOT_PASSWORD=root -p 3306:3306 -d mysql
```

E nas próximas vezes, para ativar o contêiner:

```
$ docker start meu-mysql
```

Ou parar o contêiner:

```
$ docker stop meu-mysql
```

Com o contêiner ativo, acessar o SGBD:

```
$ docker exec -it meu-mysql mysql -p
```

Usar a senha 'root' para entrar no SGBD e criar a base:

```
create database demo;
```

Sair do MySQL:

```
exit
```

Agora a mágica acontece em dois passos, primeiro passo instalar o conector do MySQL:

```
$ npm install sails-mysql --save --save-exact
```

Segundo passo modificar o arquivo */config/datastores.js*:

```
1 module.exports.datastores = {
2   default: {
3     adapter: 'sails-mysql',
4     url: 'mysql://root:root@localhost:3306/demo',
5   },
6 };
```

E pronto! Basta executar o Sails:

```
$ sails lift
```

Todo o serviço de criação das tabelas foi realizado. Podemos através do Postman criar, alterar, ver e eliminar registros no banco. Ou seja com pouquíssimas linhas de código criamos a tabela (sem entender nada de SQL) e um CRUD completo em REST (sem colocar uma única linha no controle).

4.1 Acertar a Controller

Já vimos que para disponibilizar serviços REST e gerenciar nossas estruturas de dados não precisamos fazer basicamente nada (e todo esse trabalho fica a cargo do Sails). Vamos adicionar novas funcionalidades a camada controle apenas quando necessitamos de algo especial, como por exemplo, retornar um conjunto de dados específico que desejemos.

Para entender como procedemos uma nova saída através de uma consulta SQL que agrupe os dados.

Adicionar uma nova API (Model e Controller):
`$ sails generate api Conta`

Modificar o arquivo **/api/models/Conta.js**:

```
module.exports = {
  attributes: {
    dtEntrada: {
      description: 'Data de entrada do valor na conta.',
      type: 'string',
      columnType: 'datetime',
      required: true
    },
    valor: {
      description: 'Valor entrado.',
      type: 'number',
      required: true
    }
  }
};
```

Nossa tabela de Contas tem os campos **dtEntrada** do tipo data e hora e **valor** do tipo numérico, manteremos simples assim pois o objetivo é entendermos como tudo funciona.

Realizar as alterações de permissão. No arquivo **/config/policies.js** adicionar a opção *'conta/*': true,* e no arquivo **/config/models.js** trocar a opção *migrate* para *safe*.

Ativar o Sails e criar a maior quantidade de dados que desejar variando as informações de "data de entrada" e "valor" o máximo possível. O segundo passo é adicionar um novo método de chamada no arquivo 'ContaController.js' na pasta '/api/controllers' com a seguinte codificação:

```
nomeMes = ['', 'Jan', 'Fev', 'Mar', 'Abr', 'Mai', 'Jun', 'Jul', 'Ago', 'Set', 'Out',
    'Nov', 'Dez'];

module.exports = {
  graphBase: function (req, res) {
    var myQuery = "select month(dtEntrada) as mes, sum(valor) as valor from conta " +
        "group by month(dtEntrada)";
    var meses = [];
    var valor = [];
    Conta.getDatastore().sendNativeQuery(myQuery, function (err, contas) {
      if (err) {
        return res.json({"status": 0, "error": err});
      } else {
        // sails.log(contas);
        for (var i of contas.rows) {
          meses.push(nomeMes[i.mes]);
          valor.push(i.valor);
        }
        return res.json({"mes": meses, "valor": valor});
      }
    });
  }
}
```

Selecionamos da base de dados, mês a mês o somatório dos valores e montamos para a saída um

JSON com dois arrays “mes” e “valor”, deixei o comando **sails.log** comentado caso seja necessário verificar qualquer problema na saída dos dados.

Criar uma rota para que o link **graphBase** esteja visível. No arquivo **/config/routes.js** e adicionar a seguinte linha:
`'GET /graphBase': 'ContaController.graphBase',`

Permitir a passagem desse pelas políticas, no arquivo **/config/policies.js**:
`'graphBase': true,`

Verificar se tudo funciona corretamente com o Postman:

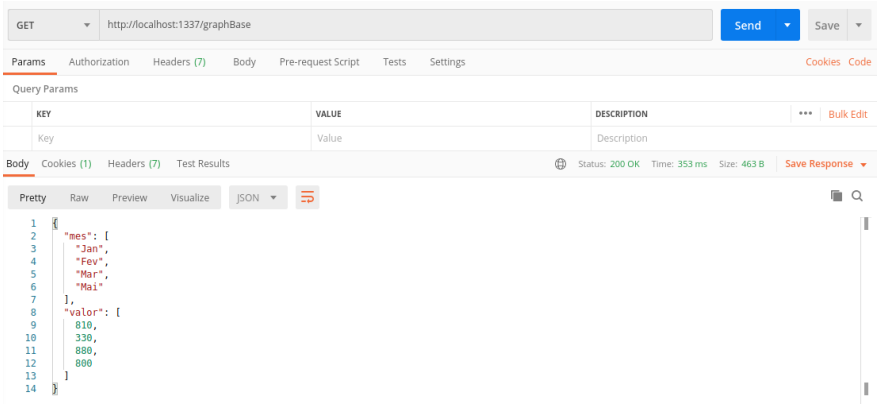

Figura 7: *Resultado da Consulta no Postman*

4.2 Visões do Projeto

A parte visual é bem simples, requer o mesmo conhecimento JavaScript ou HTML se estivéssemos programando com qualquer outro *framework*. No arquivo **/views/layouts/layout.ejs**, puxe o bloco <!–SCRIPTS–>para o header, esse bloco contém todos os JavaScripts necessários (os comentários são importantes NÃO APAGUE-OS).

Agora substituir todo o bloco BODY para:

```
<body>
  <nav class="navbar navbar-expand-lg navbar-dark bg-dark">
    <a class="navbar-brand" href="#">Meu Dashboard</a>
    <button class="navbar-toggler" type="button" data-toggle="collapse"
    data-target="#navbarTogglerDemo02" aria-controls="navbarTogglerDemo02"
    aria-expanded="false" aria-label="Toggle navigation">
      <span class="navbar-toggler-icon"></span>
    </button>
    <div class="collapse navbar-collapse" id="navbarTogglerDemo02">
      <ul class="navbar-nav mr-auto mt-2 mt-lg-0">
        <li class="nav-item active">
          <a class="nav-link" href="#">Home <span class="sr-only">(current)</span></a>
        </li>
        <li class="nav-item">
          <a class="nav-link" href="#">Link 1</a>
        </li>
        <li class="nav-item">
          <a class="nav-link disabled" href="#">Link Desabilitado</a>
        </li>
```

```
      </ul>
      <form class="form-inline my-2 my-lg-0">
        <input class="form-control mr-sm-2" type="search" placeholder="Search">
        <button class="btn btn-outline-success my-2 my-sm-0"
    type="submit">Pesquisar</button>
      </form>
    </div>
  </nav>
  <%- body %>
  <div class="container">
    <hr />
    <footer class="footer">
      <div class="pull-right">
        <a href="http://sailsjs.com">sails.js</a>
        <div>Construído com o Sails</div>
      </div>
    </footer>
  </div>
</body>
```

Definimos aqui um cabeçalho e rodapé padrão para todas as páginas do projeto. Modificar completamente o conteúdo do arquivo **/views/pages/homepage.ejs** para:

```
<div class="jumbotron">
  <div class="container">
    <h2>Início do Dashboard em Sails</h2>
    <p>Verificar se tudo está funcionando corretamente...</p>
  </div>
</div>
```

A página principal agora possui a seguinte aparência:

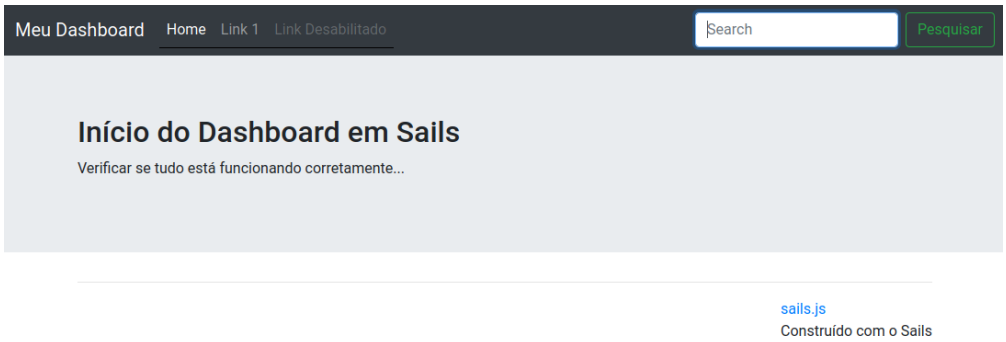

Figura 8: *Página inicial do Dashboard*

Basicamente um arquivo EJS é um arquivo HTML puro. Ou seja, do mesmo modo que programamos em qualquer sistema podemos aproveitar aqui com a vantagem de termos um MVC completo por trás.

5 Nosso Gráfico

O Sails é simplesmente incrível, ainda não me acredita! Já existe uma série de JS habilitados para o projeto, entre eles: Bootstrap 4, JQuery, Vue.js, Cloud.js e Lodash.js. Mas não temos nenhum gráfico, para incluir, por exemplo o **highcharts.js** [6] basta colocar o arquivo JS na pasta **assets/-dependencies/** e pronto. Como por passe de mágica se abrir no arquivo **/views/layout.ejs** já está a linha com a chamada a esse arquivo:

```
<script src="/dependencies/highcharts.js"></script>
```

Esse processo se repetirá para qualquer JS que incorporemos ao projeto, mesmo os particulares. Embaixo da pasta **assets/js/pages** criar uma nova pasta chamada **graficos**. E nesta adicionar um arquivo chamado **primeiro.js** com o seguinte conteúdo:

```
function criarGrafico1(meses, valores) {
  var chart = $('#grafico1').highcharts({
    chart: { type: 'bar' },
    title: { text: 'Movimento Mensal das Contas' },
    subtitle: { text: 'Ano 2020' },
    xAxis: {
      categories: meses,
      title: { text: null }
    },
    yAxis: {
      min: 0,
      title: { text: 'Valor (R$)', align: 'high' },
      labels: { overflow: 'justify' }
    },
    tooltip: { valuePrefix: 'R$', valueSuffix: ',00'
    },
    plotOptions: {
      bar: { dataLabels: { enabled: false } }
    },
    legend: {
      layout: 'vertical',
      align: 'right',
      verticalAlign: 'top',
      x: -40, y: 80,
      floating: true,
      borderWidth: 1,
      backgroundColor: Highcharts.defaultOptions.legend.backgroundColor || '#FFFFFF',
      shadow: true
    },
    credits: {
      enabled: false
    },
    series: [{
      name: 'Cliente X',
      data: valores
    }]
  });
};

$.ajax({
  type: "GET",
  url: "./graphBase",
```

```
  data: JSON.stringify({
    mes: 'mes',
    valor: 'valor'
  }),
  contentType: "application/json; charset=utf-8",
  dataType: "json",
  async: true,
  success: function (data) {
    criarGrafico1(data.mes, data.valor);
  },
  failure: function (response) {
    var r = jQuery.parseJSON(response.responseText);
    alert("Mensagem: " + r.Message);
    alert("StackTrace: " + r.StackTrace);
    alert("Tipo: " + r.ExceptionType);
  }
});
```

Se observarmos no arquivo **/views/layout.ejs** foi incorporado esse script. A primeira parte cria um gráfico conforme as instruções do próprio Highcharts (no caso um gráfico de barras horizontais) a única observação é quanto aos valores que são carregados na opção **series**.

A segunda parte é o coração de tudo, nela via AJAX vamos a URL disponibilizada através do método GET, obtemos os dados e todo esse será um processo assíncrono. Uma vez que temos os dados entraremos no evento **success** e criamos o gráfico. Caso contrário o evento **failure** será chamado e devemos verificar qual foi o problema.

Agora é questão de modificar a página principal para mostrarmos o resultado (o arquivo **/views/-pages/homepage.ejs**):

```
<div class="jumbotron">
  <figure class="highcharts-figure">
    <div id="grafico1"></div>
  </figure>
</div>
```

E o nosso resultado final é o seguinte:

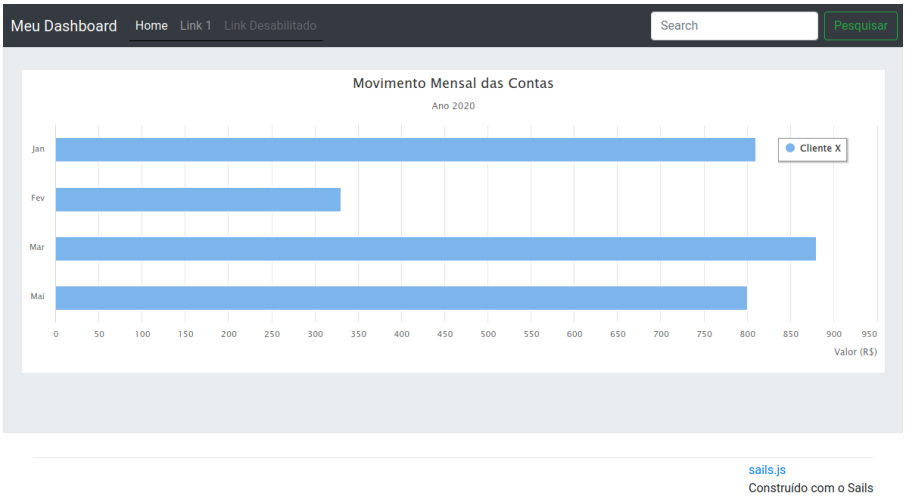

Figura 9: *Resultado Final do Dashboard*

5.1 A partir de agora

Não considere essa apostila como um passo final no aprendizado do Sails, ao contrário é o inicial, existem várias mudanças que podemos fazer nesse projeto, entre elas:

- Mudança do banco de dados, quando os dados crescerem para o Postgres ou MongoDB com o uso de drivers nativos, repectivamente *sails-postgresql* e *sails-mongo*.
- Vinculo entre a tabelas cliente a contas.
- Modificação da tabela contas para conter as movimentações de entradas e saídas.
- Limpeza dos arquivos não utilizados, o Sails já vem totalmente preparado para conter um site seguro com usuários, proteções, páginas exemplo e muitas outras funcionalidades que podemos ir incorporando ou retirando de modo a tornar nosso sistema mais veloz.

Ou seja, pensemos que agora conseguimos gerar um *Dashboard*, no caso um gerenciador de contas que podemos personalizar, criar a nossa maneira e ainda expandir isso de uma forma exponencial.

Algumas dicas finais:

- Se for publicar em um repositório de arquivos ou levar para algum outro lugar elimine a pasta **node_modules** ela é gerada com o comando `npm i`
- Os arquivos e pastas iniciados com "."possuem esse padrão pois em sistemas Unix são escondidos por padrão.
- Não se preocupe se errou algo todos os fontes deste projeto estão disponibilizados em `https://github.com/fernandoans/tstSails`
- Pessoalmente darei continuidade neste projeto e essa pode ser vista em `https://github.com/fernandoans/tstSails`

6 Conclusão

Sails é um framework JavaScript que facilita a criação de aplicativos com o servidor Node.js a nível empresarial fornece um monte de recursos poderosos por padrão, para que possamos começar a desenvolver um aplicativo sem ter que pensar sobre a configuração. Foi projetado com base no conhecido padrão MVC e com um suporte aos requisitos de aplicativos modernos: APIs orientadas a dados com uma arquitetura escalonável orientada a serviços. É possível usá-lo para qualquer projeto de aplicativo da Web.

Um desenvolvedor que possua experiência com aplicações *frontend* e está a procura de um servidor ágil de JavaScript, pode encontrar no Sails uma boa solução. Atende também aos que possuem experiência com aplicações *backend* em um idioma diferente de JavaScript e deseja expandir seus conhecimentos em Node.js. Em ambos os casos, a familiaridade com Serviços Web pode ser o quesito mais importante sobre como construir uma aplicação Web.

Um conjunto de pequenos módulos trabalham unidos no Sails para fornecer simplicidade, facilidade de manutenção e convenções estruturais aos aplicativos Node.js. Além disso é altamente configurável, assim não seremos forçados a manter toda uma funcionalidade que não é necessária para o projeto.

Sou um entusiasta do mundo **Open Source** e novas tecnologias. Qual a diferença entre Livre e Open Source? Livre significa que esta apostila é gratuita e pode ser compartilhada a vontade. Open Source além de livre todos os arquivos que permitem a geração desta (chamados de arquivos fontes) devem ser disponibilizados para que qualquer pessoa possa modificar ao seu prazer, gerar novas, complementar ou fazer o que quiser. Os fontes da apostila (que foi produzida com o LaTex) está disponibilizado no GitHub [9]. Veja ainda outros artigos que publico sobre tecnologia através do meu Blog Oficial [7].

Referências

[1] Página do Sails.js
`http://sailsjs.com`

[2] Página do Node.js
`https://nodejs.org`

[3] Página do NPM
`https://www.npmjs.com`

[4] Página do Postman
`https://www.postman.com`

[5] Página do Waterline
`https://waterlinejs.org/`

[6] Página do Highcharts
`https://www.highcharts.com`

[7] Fernando Anselmo - Blog Oficial de Tecnologia
`http://www.fernandoanselmo.blogspot.com.br/`

[8] Encontre essa e outras publicações em
`https://cetrex.academia.edu/FernandoAnselmo`

[9] Repositório para os fontes da apostila
`https://github.com/fernandoans/publicacoes`