03

Gráficos interativos com Primefaces

Começando deste ponto? Você pode fazer o DOWNLOAD (https://s3.amazonaws.com/caelum-online-public/laboratorio-java2/cap3.zip) do projeto completo do capítulo anterior e continuar seus estudos a partir deste capítulo.

Gráficos interativos com Primefaces

Por que usar gráficos?

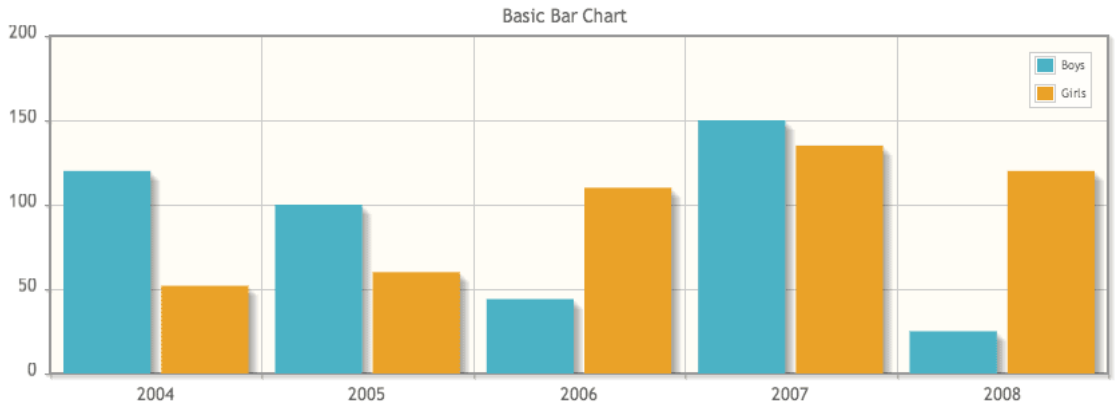
Nossa aplicação apresenta os dados das negociações tabularmente através do componente `p:dataTable`. Essa forma de mostrar informações é interessante para analisarmos dados um a um, mas não ajudam muito quando queremos ter uma ideia do que acontece com dados coletivamente.

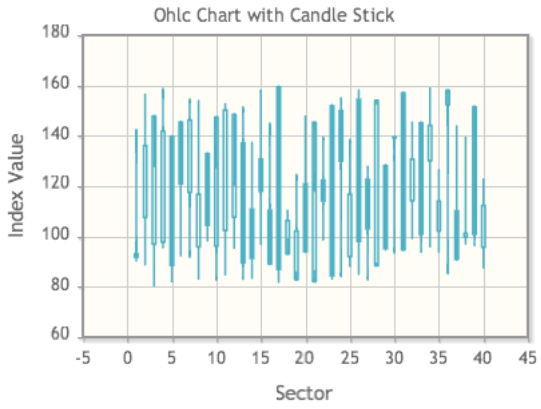
Gráficos comprovadamente ajudam no entendimento mais abrangente dos dados e são mais fáceis de analisar do que números dentro de uma tabela. É simples reconhecer padrões de imagens, por exemplo na análise técnica de valores da bolsa.

Para o projeto `Argentum`, apresentaremos os valores da `SerieTemporal` em um gráfico de linha, aplicando algum indicador como abertura ou fechamento. Continuaremos com a biblioteca Primefaces que já vem com suporte para vários tipos de gráficos.

Exemplos de gráficos

O Primefaces já possui diversos componentes para gráficos: é possível utilizá-lo para desenhar gráficos de linha, de barra, de pizza, de área, gráficos para atualização dinâmica e até para Candles, entre outros. Também é possível exportar e animar gráficos.





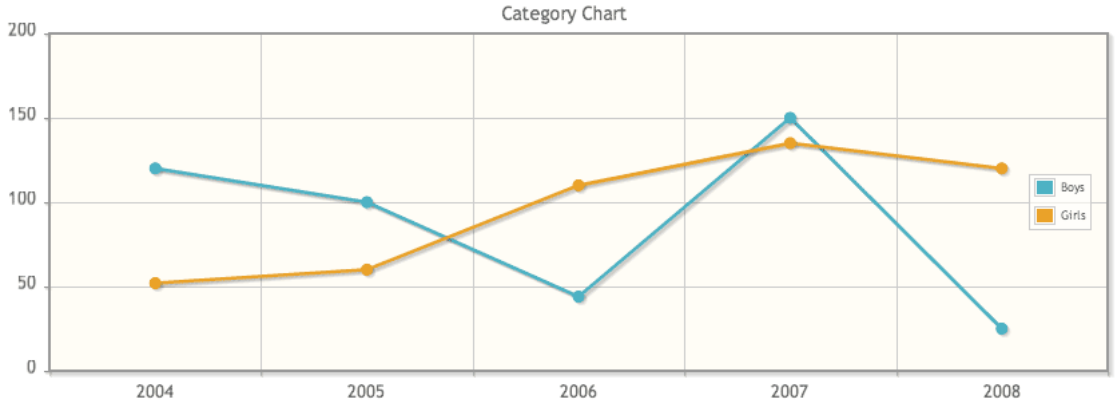
Gráficos com o Primefaces

Vamos usar o Primefaces para gerar um gráfico que mostra a evolução dos valores da série.

Nosso projeto está configurado e já podemos decidir qual gráfico utilizar. Para facilitar a decisão e ao mesmo tempo ver as possibilidades e tipos de gráficos disponíveis, o showcase do Primefaces nos ajudará muito:

http://www.primefaces.org/showcase/index.xhtml (http://www.primefaces.org/showcase/index.xhtml)

Nele encontramos o resultado final e também o código utilizado para a renderização. Vamos programar usando o componente `p:chart` que deve mostrar os valores de abertura ou de fechamento da `SerieTemporal`.



O uso de componente é simples, veja o código de exemplo do showcase:

```
<p:chart type="line" model="#{chartView.lineModel2}" />
```

Podemos ver que o componente recebe os dados (`model`) através da *Expression Language* que chama o *Managed Bean* `#{chartView.lineModel2}`.

Documentação

A documentação do Primefaces está disponível na internet e na forma de um guia do usuário em PDF. Ela fará parte do dia-a-dia do desenvolvedor, que a consultará sempre que necessário:

http://www.primefaces.org/documentation.html (http://www.primefaces.org/documentation.html)

Também há o tradicional Javadoc disponível em: http://www.primefaces.org/docs/api/5.1/ (http://www.primefaces.org/docs/api/5.1/). Devemos usar ambos para descobrir funcionalidades e propriedades dos componentes.

No showcase também há um exemplo do uso do *Managed Bean*, porém para maiores informações é fundamental ter acesso ao Javadoc e à documentação da biblioteca para saber quais classes, atributos e métodos utilizar.

Aplicando ao nosso projeto

Para mostrarmos o gráfico no Argentum, usaremos a tag do Primefaces vista acima:

```
<p:chart type="line" />
```

Da forma como está, no entanto, não há nada a ser mostrado no gráfico. Ainda falta indicarmos para o componente que os dados, o modelo do gráfico, será disponibilizado por nosso `ManagedBean`. Em outras palavras, faltou indicarmos que o `model` desse gráfico será produzido em `argentumBean.modeloGrafico`.

Nossa adição ao `index.xhtml` será, portanto:

```
<p:chart type="line" model="#{argentumBean.modeloGrafico}" />
```

Definição do modelo do gráfico

Já temos uma noção de como renderizar gráficos através de componentes do Primefaces. O desenvolvedor não precisa se preocupar como detalhes de JavaScript, imagem ou animações. Tudo isso é encapsulado no próprio componente, seguindo boas práticas do mundo orientado a objetos.

Apenas, será necessário informar ao componente quais são os dados a serem plotados no gráfico em questão. Esses dados representam o modelo do gráfico e devem ser disponibilizados como o `model` para o `p:chart` em um objeto do tipo `org.primefaces.model.chart.ChartModel`.

Essa é a classe principal do modelo e há classes filhas especializadas dela como `LineChartModel`, `PieChartModel` ou `BubbleChartModel`. No JavaDoc podemos ver todas as filhas da `ChartModel` e ainda no *showcase* é possível ver o `ManagedBean` responsável por cada modelo de gráfico. Assim, saberemos qual delas devemos usar de acordo com o gráfico escolhido:

Javadoc: http://www.primefaces.org/docs/api/5.1/org/primefaces/model/chart/ChartModel.html (http://www.primefaces.org/docs/api/5.1/org/primefaces/model/chart/ChartModel.html)

No nosso projeto, utilizaremos o `LineChartModel`, já que queremos plotar pontos em um gráfico de linha. Um `LineChartModel` recebe uma ou mais `ChartSeries` e cada `ChartSeries` representa uma linha no gráfico do componente `p:chart`. Uma `ChartSeries`, por sua vez, contém todos os pontos de uma linha do gráfico, isto é, os valores X e Y que serão ligados pela linha do gráfico.

Vejo como fica o código fácil de usar:

```
LineChartSeries serieGrafico = new LineChartSeries();
serieGrafico.set("dia 1", 20.9);
```

```
serieGrafico.set("dia 2", 25.1);
serieGrafico.set("dia 3", 22.6);
serieGrafico.set("dia 4", 24.6);

LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(serieGrafico);
```

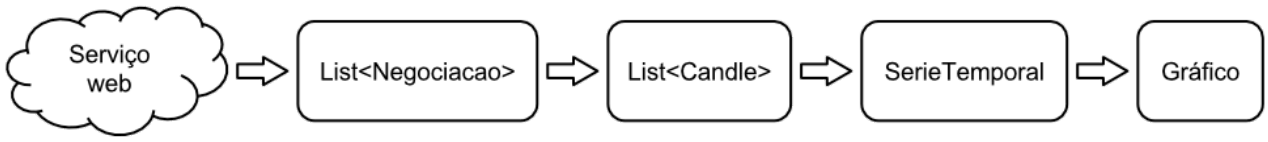
Uma `ChartSeries` recebe no construtor a legenda da linha que ela representa (*label*) e, através do método `set`, passamos os valores de cada ponto nos eixos horizontal e vertical, respectivamente. No exemplo acima, os valores colocados são fixos, mas na nossa implementação do Argentum, é claro, iteraremos pela `SeriaTemporal` calculando os indicadores sobre ela.

Isto é, uma vez que temos a `SerieTemporal`, nosso código para plotar a média móvel simples do fechamento em uma `ChartSeries` será semelhante a este:

```
SerieTemporal serie = ...
LineChartSeries linha = new LineChartSeries();
linha.setLabel("MMS - Fechamento");
MediaMovelSimples indicador = new MediaMovelSimples();
for (int i = 2; i < serie.getUltimaPosicao(); i++) {
    double valor = indicador.calcula(i, serie);
    chartSeries.set(i, valor);
}

LineChartModel modeloGrafico = new LineChartModel();
modeloGrafico.addSeries(chartSeries);
```

Conseguir a série temporal é um problema pelo qual já passamos antes. Relembre que a `SerieTemporal` é apenas um *wrapper* de uma lista de `Candles`. E a lista de `Candles` é gerada pelo `CandlestickFactory` resumindo uma lista com muitas `Negociacoes`.



Se procurarmos o local no nosso código onde pegamos a lista de negociações do web service, vamos notar que isso acontece no construtor da `ArgentumBean`. Seu código completo ficaria assim;

```
public ArgentumBean() {
    this.negociacoes = new ClienteWebService().getNegociacoes();
    List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
    SerieTemporal serie = new SerieTemporal(candles);

    LineChartSeries linha = new LineChartSeries();
    linha.setLabel("MMS - Fechamento");
    MediaMovelSimples indicador = new MediaMovelSimples();
    for (int i = 2; i < serie.getUltimaPosicao(); i++) {
        double valor = indicador.calcula(i, serie);
        chartSeries.set(i, valor);
    }
```

```
        this.modeloGrafico = new LineChartModel();
        modeloGrafico.addSeries(linha);
    }
```

Note, no entanto, que esse trecho de código está com responsabilidades demais: ele busca as negociações no *web service*, cria a série temporal, plota um indicador e disponibiliza o modelo do gráfico. E, pior ainda, esse código ainda está no construtor do `ArgentumBean` !

Com todo esse código no construtor do bean, teríamos um código mais sujo, pouco coeso e muito difícil de testar. O que acontece aqui é que não estamos separando responsabilidades o bastante e nem encapsulando a lógica de geração de gráfico corretamente.

Isolando a API do Primefaces: baixo acoplamento

O que acontecerá se precisarmos criar dois gráficos de indicadores diferentes? Vamos copiar e colar todo aquele código e modificar apenas as partes que mudam? E se precisarmos alterar algo na geração do modelo? Essas mudanças não serão fáceis se tivermos o código todo espalhado pelo nosso programa.

Os princípios de orientação a objetos e as boas práticas de programação vêm ao nosso socorro aqui. Vamos encapsular a maneira como o modelo do gráfico é criado na classe `GeradorModeloGrafico` .

Essa classe deve ser capaz de gerar o `ChartModel` para nosso gráfico de linhas, com os pontos plotados pelos indicadores com base nos valores de uma série temporal. Isto é, nosso `GeradorModeloGrafico` precisa receber a `SerieTemporal` sobre a qual plotar os indicadores.

Se quisermos restringir o gráfico a um período menor do que o devolvido pelo *web service*, é uma boa recebermos as posições de início e fim que devem ser plotadas. Esse intervalo também serve para que não tentemos calcular a média da posição zero, por exemplo -- note que, como a média é dos três últimos valores, tentaríamos pegar as posições `0` , `-1` e `-2` , o que causaria uma `IndexOutOfBoundsException` .

Como essas informações são fundamentais para qualquer gráfico que plotemos, o gerador do modelo do gráfico receberá tais informações no construtor. Além delas, teremos também o objeto do `LineChartModel` , que guardará as informações do gráfico.

```
public class GeradorModeloGrafico {

    private SerieTemporal serie;
    private int comeco;
    private int fim;
    private LineChartModel modeloGrafico;

    public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
        this.serie = serie;
        this.comeco = comeco;
        this.fim = fim;
        this.modeloGrafico = new LineChartModel();
    }

}
```

E, quem for usar essa classe, fará:

```
SerieTemporal serie = //...
GeradorModeloGrafico g = new GeradorModeloGrafico(serie, inicio, fim);
```

Repare como o código que usa o `GeradorModeloGrafico` não possui nada que o ligue ao `LineChartModel` especificamente. O dia em que precisarmos mudar o gráfico a ser plotado ou mesmo mudar a tecnologia que gerará o gráfico, só precisaremos alterar a classe `GeradorModeloGrafico` . Relembre o conceito: esse é o poder do **encapsulamento**!

E nossa classe não se limitará a isso: ela encapsulará tudo o que for relacionado ao gráfico. Por exemplo, para criar o gráfico, precisamos ainda de um método que plote os pontos calculados por um indicador, como o `plotaMediaMovelSimples` abaixo. Ele passa por cada posição do `comeco` ao `fim` da `serie` , chamando o cálculo da média móvel simples.

```
public void plotaMediaMovelSimples() {
    MediaMovelSimples indicador = new MediaMovelSimples();
    LineChartSeries linha = new LineChartSeries();
    linha.setLabel("MMS - Fechamento")

    for (int i = comeco; i <= fim; i++) {
        double valor = indicador.calcula(i, serie);
        linha.set(i, valor);
    }
    this.modeloGrafico.addSeries(linha);
}
```

O método `plotaMediaMovelSimples` cria um objeto da `MediaMovelSimples` e varre a `SerieTemporal` recebida para calcular o conjunto de dados para o modelo do gráfico.

Podemos aproveitar também para adicionar a configuração do nome do gráfico e da posição da legenda no nosso construtor:

```
public GeradorModeloGrafico(SerieTemporal serie, int comeco, int fim) {
    this.serie = serie;
    this.comeco = comeco;
    this.fim = fim;
    this.modeloGrafico = new LineChartModel();

    this.modeloGrafico.setTitle("Indicadores");
    this.modeloGrafico.setLegendPosition("w");
}
```

Por fim, ainda precisaremos de um método `getModeloGrafico` que devolverá o modelo para o `ArgentumBean` , já que o componente `p:chart` é que precisará desse objeto preenchido. Repare que o retorno é do tipo `ChartModel` , super classe do `LineChartModel` . É boa prática deixar nossa classe a mais genérica possível para funcionar com qualquer tipo de método.

Veja como fica o programa dentro do *ArgentumBean* e note que essa classe apenas delega para a outra toda a parte de criação do gráfico:

```
public class ArgentumBean {

    private List<Negociacao> negociacoes;
    private String indicadorBase;
    private String media;
    private ChartModel modeloGrafico;

    public ArgentumBean() {
        this.negociacoes = new ClienteWebService().getNegociacoes();
        List<Candle> candles = new CandleFactory().constroiCandles(negociacoes);
        SerieTemporal serie = new SerieTemporal(candles);

        GeradorModeloGrafico geradorGrafico =
                    new GeradorModeloGrafico(serie, 2, serie.getUltimaPosicao());
        geradorGrafico.plotaMediaMovelSimples();
        this.modeloGrafico = geradorGrafico.getModeloGrafico();
    }

    public List<Negociacao> getNegociacoes() {
        return negociacoes;
    }

    public ChartModel getModeloGrafico() {
        return modeloGrafico;
    }
}
```

Note que, para quem usa o `GeradorModeloGrafico` nem dá para saber como ele gera o modelo. É um código **encapsulado**, **flexível**, **pouco acoplado** e **elegante**: usa boas práticas da Orientação a Objetos.

O que aprendemos:

- Como utilizar o componente de gráficos do primefaces.
- Configurar títulos, legendas e sua posição nos gráficos.
- A separar e as responsabilidades de cada classe.
- Fazer um código encapsulado e pouco acoplado.
- Como plotar o gráfico da Média Móvel Simples.