

MM - Camadas da Arquitetura

Camadas da Arquitetura

Anteriormente na seção *Proposta da Arquitetura*, abordamos a interação entre componentes de cada uma das camadas. Além da interação, também introduzimos os frameworks utilizados nesta camada.

Camada de Visualização

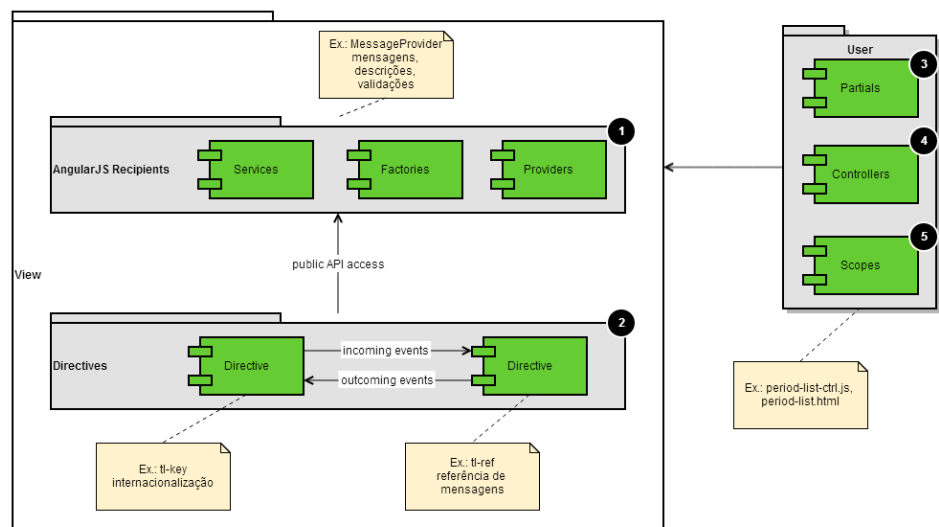
Conforme citado na seção de contextualização, a arquitetura proposta visa eliminar os problemas identificados em diversos projetos já executados, onde foi possível identificar a falta de padronização para realizar tarefas comuns no processo de desenvolvimento. Assim, temos como objetivo principal desta seção o detalhamento do projeto Commons UI que provê soluções para que seja possível estabilizar, o que era previamente uma atividade caótica, em uma atividade produtiva e padronizada. Em busca deste equilíbrio, foram estudadas soluções e padrões de mercado que balizam o desenvolvimento de software atualmente.

O framework AngularJS foi a opção escolhida após compararmos as soluções disponíveis. A possibilidade de extensão do framework nos proporciona a flexibilidade necessária para criar componentes personalizáveis que atendem plenamente as nossas necessidades. Além do benefício da extensibilidade, o framework possui uma documentação detalhada, comunidade participativa e componentes genéricos já desenvolvidos e inclusos na solução. Para maiores detalhes, consulte o guia para desenvolvedores do [AngularJS](#).

Tomando como base os motivos citados acima, criamos o projeto commons-ui. Este projeto compõe a camada de apresentação. Ao iniciar um projeto é possível ter o commons-ui como dependência para ter acesso a suas funcionalidades e compartilhar a padronização através de componentes como: directives, services, providers, factories, controllers, scopes, partials, etc.

Esta seção será dividida em três grandes tópicos: *estrutura*, *detalhamento da estrutura* e *melhorias futuras*. O primeiro tópico aborda a organização do projeto e responsabilidade de cada componente, descrevendo brevemente o comportamento dos componentes que compõem a estrutura do projeto. O tópico de detalhamento mostra detalhadamente o funcionamento dos componentes, a comunicação entre diferentes componentes e sua utilização pelo desenvolvedor. E por fim, o tópico de melhorias, no qual irá listar diversas melhorias que podem ser realizadas pela equipe de arquitetura para facilitar na construção de novos componentes e manter a atual estrutura da arquitetura estável.

O diagrama abaixo ilustra, de forma macro, a comunicação e organização dos componentes na atual versão da arquitetura.



Itens	Nome	Descrição
1	Recipients	Services , factories e providers são componentes disponibilizados pela arquitetura que expõem métodos públicos e são passíveis de injeção de dependências.
2	Directives	Directives são como uma extensão do HTML na forma de atributos anexados nestes elementos. Diretivas utilizam-se de services, factories, providers e eventos para comunicar-se e realizar suas funções.
3	Partials	Partials são páginas HTML que podem ser incorporadas em outras páginas através da directive <i>ng-include</i> . Todas as páginas de listagem e detalhe da arquitetura são partials que compoem a estrutura HTML da aplicação.
4	Controllers	Controllers interagem com as partials e controlam itens de escopo e diretivas. Métodos e regras de negócio são expostos para as partials através de controllers.

5	Scopes	Scopes formam a cola entre controllers e directives. Para expor um método para uma diretiva, este método deve estar declarado dentro de um escopo. Cada controller possui um escopo.
---	--------	--

Recipients

Recipients é um termo utilizado na documentação do framework AngularJS para referir-se a um agrupador contendo os componentes: service, factory e provider. Os componentes deste agrupador são abordado no decorrer desta seção.

O que é um Service?



Atenção

A arquitetura não faz uso de services, apenas factories e providers. Apesar de não ser utilizado internamente na arquitetura, registrar e fazer uso de services é um conceito importante que pode ser aplicado nesta camada.

São objetos que compartilham funcionalidades através de APIs públicas para toda aplicação que faz uso do AngularJS. Services fornecem injeção de dependência e padronização quanto a criação de APIs públicas.

Services são registrados a partir da [Module API](#). Ao registrar um service, dá-se um nome no qual será utilizado para injeção de dependências em quaisquer outros componentes do AngularJS. Abaixo temos um exemplo da utilização de services para manter estados de um dialog.

Registrando e Utilizando Services

```
// aqui o service com nome DateUtil (primeiro parâmetro na chamada da função service()) é registrado no módulo
treelayer.common.ui.util
angular.module('treelayer.common.ui.util').service( 'DateUtil', function() {

    this.format = function( date ) {
        new Date(date).format('dd-mm-yyyy');
    };
});

// em um modulo que depende do commons (shieldApp no caso da aplicação Shield) o componente DateUtil é injetado
// em um controller através do array de string (segundo argumento da função controller()) onde este argumento
// deve conter o nome que o componente foi registrado, ou seja, 'DateUtil'
shieldApp.controller( 'PeriodController', [ 'DateUtil', '$scope', function( DateUtil, scope ) {

    // ...

    // metodo chamado para formatar a data escolhida em um widget calendar
    scope.formatCurrentDate = function(date) {
        scope.formattedDate = DateUtil.format(date); // utilização do service DateUtil
    };

}]);
```

Note a forma que declaramos a função format no service acima. Internamente, ao declarar um service, a função retornada será instanciada apenas quando necessário (lazy). Esta instanciação dá-se chamando construtor da função através do operador new. Services são estruturas simples, expõem métodos declarados no contexto (variável implícita this) e serão instanciados pelo AngularJS a partir de um construtor padrão. São utilizados quando não há necessidade de manter estado nos objetos e/ou criar objetos que necessitam de parâmetros para sua instanciação.

O que é um Factory?

Assim como o [service](#), factories também são objetos que compartilham interfaces públicas de API e são passíveis a injeção de dependências.

Factories são registradas através da [Module API](#). Ao registrar uma factory, dá-se um nome no qual será utilizado para injeção de dependências em quaisquer outros componentes do AngularJS. Abaixo temos um exemplo de como registrar uma factory e como utiliza-la através da injeção de dependências.

Registrando e Utilizando Factories

```
// aqui a factory com nome $dialog(primeiro parâmetro na chamada da função service()) é registrada no módulo
treelayer.commons.ui
angular.module('treelayer.commons.ui').factory('$dialog', function() {

    var Dialog = function(isOpen, operation) {
        this.isOpen = isOpen ? isOpen : false;
        this.operation = operation ? operation : '';
    };

    Dialog.prototype.close = function() {
        this.isOpen = false;
        this.operation = '';
    };

    Dialog.prototype.open = function(op) {
        this.isOpen = true;
        if (op) this.operation = op;
    };

    return {
        'new': function(isOpen, operation) {
            return new Dialog(isOpen, operation);
        }
    };
});

// em um modulo que depende do commons (shieldModule no caso da aplicação Shield) o componente $dialog é
injetado
// em um controller através do array de string (segundo argumento da função controller()) onde este argumento
// deve conter o nome que o componente foi registrado, ou seja, '$dialog'
shieldApp.controller( 'PeriodCtrl', ['$dialog', '$scope', function ( Dialog, scope ) {

    // ...
    $scope.view = {
        detailDialog: Dialog.new() // utilização da factory $dialog
    };

    // ...

}]);
```

Note a forma que declaramos a função `foo` na factory acima. Internamente, ao declarar uma factory, a função `foo` já está instanciada e apenas será executada, diferentemente do `service` onde uma nova função será instanciada através do operador `new`. A factory `$dialog` oferece um método público `new` que constrói uma nova instância do objeto `Dialog`. Factories fornecem mais flexibilidade na sua instanciação, possibilitando a construção de objetos que possuam parametro, por isso, em casos como mostrado no exemplo acima são preferíveis ao invés de `services`.



Código Atualizado

Acesse o código-fonte da factory `$dialog` e do controller `PeriodCtrl`.

O que é um Provider?

Providers, assim como `services` e `factories` provem interfaces públicas e são passíveis de dependência. Diferentemente destes outros componentes, providers são expostos antes da inicialização da aplicação. Este atributo fornece aos providers um mecanismo de configuração, tornando-os úteis onde o comportamento de seus métodos públicos diferem conforme a configuração realizada em um bloco de inicialização do AngularJS.

São registrados através da [Module API](#). Ao registrar um provider, dá-se um nome (primeiro parâmetro na chamada de função `angular.module('treelayer.commons.ui').provider()` no qual será utilizado para injeção de dependências em qualquer outro componente do AngularJS. Abaixo temos um exemplo de como registrar um provider e como utiliza-lo através da injeção de dependências.

Registrando e Utilizando Providers

```
angular.module('treelayer.common.ui').provider('$message', function () {  
  
    // ...  
    this.baseUrl = '';  
  
    // ...  
  
    this.$get = ['$q', '$http', function ($q, $http) {  
        return this.$message($q, $http);  
    }];  
  
    // ...  
});
```

Todo provider deve possuir um método `$get`. Este será invocado pelo AngularJS para instanciar o provider e deixá-lo disponível para o bloco de configuração da aplicação. Internamente, `services` e `factories` são instâncias de providers, onde o método `$get` é injetado automaticamente pelo framework para [facilitar o desenvolvimento](#).

A injeção de dependências de providers muda conforme a localidade que estão sendo injetados. Como podemos perceber no bloco de código acima, o provider de exemplo foi registrado com o nome `fooProvider`. Para injetarmos providers em controllers, retira-se o prefixo `provider`, restando apenas `$message`, como pode ser visto no bloco de configuração da aplicação Shield. Em blocos de configuração, mantém-se o prefixo como no bloco de exemplo abaixo.

Utilizando Providers em Blocos de Configuração

```
shieldApp.config(['$messageProvider',  
    '$routeProvider',  
    'partialsPath',  
    function ($messageProvider, $routeProvider, partialsPath) {  
  
        // ...  
  
        $messageProvider.baseUrl = AJS.contextPath() + '/rest/shield/latest';  
  
        // ...  
    }]);
```



Código Atualizado

Acesse o código-fonte do provider [\\$messageProvider](#) e do [bloco de inicialização da aplicação Shield](#).

O que é uma Directive?

Diretivas são marcadores (podem ser atributos, classes ou novos elementos) em nodos da árvore DOM, como uma extensão da especificação construída pela [W3C](#). Em sua inicialização, o framework AngularJS percorre a árvore HTML, compila e executa as directives declaradas. Durante sua execução, directives são capazes de anexar novos comportamentos e funcionalidades que não existiam, até então, na estrutura especificada pela W3C.

O framework [AngularJS disponibiliza, por padrão, um conjunto de directives](#). Assim como você pode criar controllers e services, directives também podem ser customizadas. A partir desta extensibilidade foram criadas directives que atendem as necessidades de padronização, como por exemplo a directive `tl-key`.

Abaixo temos um exemplo da utilização da directive `tl-key` e `ng-click` que ilustra como utilizar a directive customizada e uma directive empacotada com o AngularJS.

Utilizando Directives - Partial

```
<!-- ... -->

<div class="inline-field-container" tl-ref="common">
  <button class="aui-button align-right" ng-click="view.detailDialog.open('create')" tl-key="button.add"><
/ button>
</div>

<!-- ... -->
```

Utilizando Directives - Controller

```
shieldApp.controller( 'PeriodCtrl', ['$scope', '$http', '$property', '$resourceLocator', '$dialog', function (
$scope, $http, $property, $resourceLocator, $dialog ) {

    // ...

    $scope.view = {

        // ...

        detailDialog: $dialog.new()

        // ...

    };

    // ...

}]);
```

No primeiro blocos acima temos o exemplo do uso da directive `tl-key` e `ng-click`; um partial, de onde retiramos um fragmento do HTML. A directive `tl-key`, insere os labels de forma internacionalizada a partir de uma chave. A directive `ng-click`, invoca o método `open` do objeto `view.detailDialog` contigo no controller `PeriodCtrl`.

Nestes exemplos podemos ver como é dada a interação entre controllers, scopes, partials e directives, onde o conjunto destes itens implica em uma interface dinâmica para o usuário final, agregando qualidade e padronização no processo de desenvolvimento.



Código Atualizado

Acesse o código-fonte do partial [period-list.html](#) e do controller [PeriodCtrl](#).

O que é um Partial?

Partials são páginas HTML que podem ser incorporadas em outras páginas através da directive `ng-include`. Todas as páginas de listagem e detalhe da arquitetura são partials que compoem a estrutura HTML da aplicação.

Abaixo temos um exemplo que da utilização da directive `ng-include` para incorporar o dialog de detalhe na página de períodos da aplicação Shield.

Listagem de Períodos - Partial

```
<!-- ... -->

    <div ng-include="view.detailDialogUrl.url" />

<!-- ... -->
```

Detalhe de Períodos - Partial

```
<div tl-object="period" tl-context="detail" tl-dialog tl-opened="$parent.view.detailDialog.isOpen">

  <!-- ... -->

  <!-- ... -->

</div>
```



Código Atualizado

Acesse o código-fonte dos parciais [period-list.html](#) e [period-detail.html](#).

O que é um Controller?

De acordo com a Wikipedia, um controller pode enviar comandos para sua visão associada para alterar a apresentação da visão do modelo (por exemplo, percorrendo um documento). Ele também pode enviar comandos para o modelo para atualizar o estado do modelo (por exemplo, editando um documento).

Adaptando a citação para o contexto do AngularJS, o controller interage com directives (presentes em elementos HTML nos parciais) através do escopo, podendo modificar o comportamento na apresentação de conteúdos ou regras para o usuário final.

O exemplo abaixo mostra como o controller `PeriodCtrl` preenche uma combo-box na partial `period-list.html` através da directive `ng-option`.

Utilizando Controllers - Partial

```
<!-- ... -->

<select class="select field-short" ng-model="model.enabled" tl-key="enabled" ng-options="key.bool() as value
for (key, value) in view.status">
  <option value=""></option>
</select>

<!-- ... -->
```

Utilizando Controllers - Controller

```
shieldApp.controller( 'PeriodCtrl', ['$scope', '$http', '$property', '$resourceLocator', '$dialog', function (
$scope, $http, $property, $resourceLocator, $dialog ) {

    // ...

    $scope.view = {

        // ...

        status: $property.range({object: 'period', attribute: 'enabled'}),

        // ...

    };

    // ...

}]);
```



Código Atualizado

Acesse o código-fonte do parcial [period-list.html](#) e do controller [PeriodCtrl](#).

O que é um Scope?

É um link entre partials e controllers, onde tudo que declaramos em scopes é exposto para ser utilizado em partials, por directives. Contudo, scopes possuem [outras funcionalidades](#), como por exemplo a função `$watch` que fornece ao desenvolvedor uma forma de observar valores contidos no scope e realizar ações conforme a interação do usuário com a UI.

Exemplo de código onde uma variável declarada em um escopo do controller `PeriodCtrl` é exposta para a directive `ng-options`.

Utilizando Controllers - Partial

```
<!-- ... -->

<select class="select field-short" ng-model="model.enabled" t1-key="enabled" ng-options="key.bool() as value
for (key, value) in view.status">
  <option value=""></option>
</select>

<!-- ... -->
```

Utilizando Controllers - Controller

```
shieldApp.controller( 'PeriodCtrl', ['$scope', '$http', '$property', '$resourceLocator', '$dialog', function (
$scope, $http, $property, $resourceLocator, $dialog ) {

    // ...

    $scope.view = {

        // ...

        status: $property.range({object: 'period', attribute: 'enabled'}),

        // ...

    };

    // ...

}]);
```

No exemplo acima, acessamos o scope do controller `PeriodCtrl` através de injeção de dependências no construtor do controller (variável declarada como `$scope`). Valores são expostos para partials e directives através do operador `.` (`dot`). No exemplo, um objeto de nome `view` é criado no scope, e internamente este objeto possui um atributo `status`. A partir daí, a partial pode referenciar o objeto `status` através da atributo `view` do scope; como vemos na declaração `view.status`.



Código Atualizado

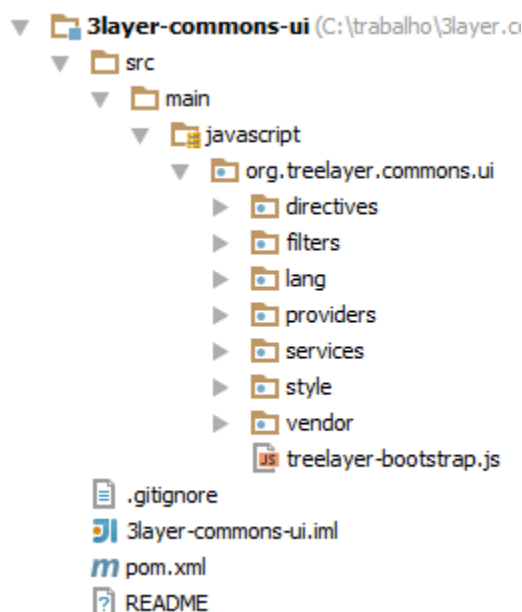
Acesse o código-fonte do partial [period-list.html](#) e do controller [PeriodCtrl](#).



Não deixe de ler a [documentação oficial](#), funcionalidades como: prototypal inheritance, listeners, observers, data-binding, etc, são importantes no desenvolvimento da camada de apresentação.

Estrutura do Projeto Commons UI

Conforme a imagem abaixo, o projeto é estruturado por pastas, cada qual refletindo a responsabilidade de seus componentes. Como este projeto é inteiramente composto de arquivos Javascript, não existem dependências de outros projetos.



Pasta	Descrição
Directives	Contém todas as diretivas customizadas que estão disponíveis para uso dos desenvolvedores. Estas diretivas foram criadas para atender os requisitos da arquitetura e não interferem nas diretivas do AngularJS.
Filters	Contém filtros customizados que estão disponíveis para uso dos desenvolvedores. Estes filtros foram criadas para atender os requisitos da arquitetura e não interferem nas diretivas do AngularJS.
Lang	Contém funções utilitárias, não disponibilizadas pela API Javascript nativa dos navegadores. Podemos citar como exemplo as funções: each e curry.
Providers	Contém providers utilizados internamente pela arquitetura para evitar alto acoplamento entre as diretivas.
Services	Contém serviços declarados através da API do AngularJS que estão disponíveis para injeção e podem ser utilizados pelos desenvolvedores para padronizar e agilizar o processo de desenvolvimento de novas funcionalidades.
Style	Contém arquivos .css para padronização de elementos de UI.
Vendor	Contém as bibliotecas previamente citadas neste documento já minificadas.

Além das pastas que agrupam os componentes por responsabilidade, o projeto possui o arquivo *treelayer-bootstrap* que provê um módulo onde estão listadas todas as dependências. Este arquivo existe para que quando necessário, projetos especifiquem como dependência apenas este módulo e todos os componentes estarão automaticamente associados ao projeto.

Providers Disponibilizados pela Arquitetura

Providers são objetos expostos para toda a aplicação, passíveis de injeção, oferecem funções públicas e podem ser utilizados dentro de um bloco de configuração do AngularJS. Para maiores informações leia a documentação oficial sobre [providers](#).

Message Provider

É o componente responsável por disponibilizar mensagens, descrições e validações e controlar o fluxo de execução das diretivas que fazem uso destes itens. Este componente é apoiado por endpoints RESTful que fornecem estas informações de forma dinâmica. Para realizar estas chamadas, em um bloco de configuração do AngularJS é possível configurar as diferentes URLs através de injeção de dependências. Abaixo temos uma imagem do projeto Shield, onde é configurado as URLs utilizadas pelo MessageProvider.


```
shieldApp.config(['$messageProvider',
    '$routeProvider',
    'partialsPath',
    function ($messageProvider, $routeProvider, partialsPath) {

        $messageProvider.baseUrl = AJS.contextPath() + '/rest/shield/latest';
        $messageProvider.baseMessageUrl = '/services/messages';
        $messageProvider.baseValidationUrl = '/services/validation';
    }
]);
```

As funções disponíveis publicamente são:

Nome	Descrição	Utilização
whenMessageLoads	Fornecer uma forma de se registrar a este provider e, quando o back-end retornar, o provider irá alertar os componentes registrados.	<pre>\$message.whenMessageLoads(self.cacheKey, function(cacheKey) { self.executeReferenceCallbacks(cacheKey); self.executeKeyCallbacks(cacheKey); });</pre>
getRawMessages	Retorna um objeto contendo todas as mensagens, validações e descrições. Este objeto não sofreu nenhuma modificação por directives, está no formato <i>raw</i> .	<pre>curriedProcessMessages(\$message.getRawMessages(\$super.cacheKey));</pre>
getValidations	Retorna um objeto contendo todas as validações.	<pre>\$message.getValidations();</pre>
createReference	Usado principalmente pela directive <i>tl-ref</i> , na qual cria o cache de referências.  Este método é público mas não deve ser utilizado, foi criado para estabelecer a comunicação entre a directive e o cache de referências.	<pre>/** * Create a reference in message reference cache. * * @param cacheKey global reference cache identifier * @param tlRef identifier inside tlRef * @param messages messages to cache * * @InternalApi */ this.createReference = function(cacheKey, tlRef, messages)</pre>
findMessagesByKey	Busca uma determinada mensagem pela chave contida no arquivo <i>property</i> .	<pre>\$message.findMessagesByKey('common.button.view');</pre>

Busca de Mensagens

A partir de uma chamada da principal directive, ***tl-object***, é disparada uma requisição HTTP para um método genérico da arquitetura para buscar mensagens internacionalizadas. Esta funcionalidade baseia-se no nome do objeto passado para a directive ***tl-object*** para buscar as mensagens. Mensagens são compostas de labels, descriptions, validations, etc.

As URLs previamente configuradas ao inicializar um aplicação AngularJS servem como endpoint para esta funcionalidade.

Armazenamento de Mensagens

Após a resposta do servidor, se ocorrida com sucesso, todas as mensagens são armazenadas em cache no formato *raw*, ou seja, sem alteração nenhuma. Assim que possível outras diretivas irão processar estas mensagens e preencher outras áreas de cache deste provider.

Controle de Fluxo

Logo após a busca de mensagens e a resposta do servidor, todas as diretivas que se registraram neste provider serão avisadas que há mensagens não processadas. A directive `tl-ref` iniciará sua execução preenchendo o cache `referenceMessages` a com uma nova estrutura, oriunda do cache no formato `raw`. Este cache é disponibilizado para outras diferitvas através da função pública exposta `findMessagesByKey`.

 Se você possui dúvidas quanto a funcionalidade das directives citadas nesta seção, leia as seções de directives *Object* e *Ref*.

Services Disponibilizados pela Arquitetura

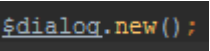
 Nesta seção, ao mencionar services estamos nos referindo a [factories](#), [services](#) e [providers](#), ou seja, toda e qualquer implementação destes componentes. Isto ocorre pois service é um nome genérico que pode ser utilizado para documentação, porém o AngularJS utiliza deste nome para componentes também.

Services são objetos compostos por funções, passíveis de injeção de dependências e inicializados. Um serviço no framework AngularJS é tratado como um singleton e pode ser injetado através de seu nome, bastando declara-lo como parametro em seus componentes (no caso, um controller, por exemplo). Para maiores informações acesse a documentação oficial sobre [services](#).

Dialog Service

É um serviço declarado através do framework AngularJS que possui unicamente a funcionalidade de retornar um objeto que controla estados de um dialog. Mencionamos na introdução de services que estes não possuem estado, porém o que citamos na anteriormente foi que o Dialog Service tem como unica funcionalidade retornar um objeto com estados. Como isto se dá? Basicamente este serviço possui uma única função, a função responsável por criar o objeto dialog, ou seja, o Dialog Service não possui estado, mas sim objeto obtido no retorno desta chamada.

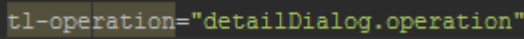
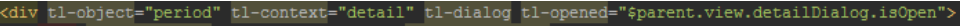
As funções disponíveis publicamente são:

Nome	Descrição	Utilização
new	Cria um objeto do tipo Dialog para que o usuário possa controlar as operações e a visibilidade de um determinado dialog.	

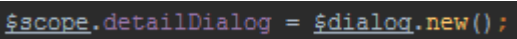
Dialog

Dialog é um objeto definido internamente no service `DialogService` e tem como responsabilidade controlar os estados de um dialog. O estado é basicamente quando o dialog está aberto e a operação que esta ocorrendo no momento. Para obter um objeto do tipo Dialog basta realizarmos uma chamada para a função `new` no `DialogService`.

Os atributos disponíveis publicamente são:

Nome	Descrição	Utilização
operation	Define se o dialog quando exibido esta no modo de leitura ou edição.	
isOpen	Variável de controle utilizada para determinar se o Dialog está visível.	

As funções disponíveis publicamente são:

Nome	Descrição	Utilização
new	Inicializa um objeto do tipo Dialog e retorna.	

open	Altera o valor da atributo isOp en para false. Esta faz com que o dialog seja exibido para o usuário. No exemplo de utilização ao lado é possível notar que podemos passar um valor do tipo String. Este valor é a operação executada no dialog, quando aberto. Os valores aceitos são: <i>create</i> , <i>update</i> , <i>delete</i> e <i>view</i> .	<pre><button class="aul-button align-right" ng-click="detailDialog.open('create')" t1-key="button.add" t1-save="model" /></pre>
close	Altera o valor da atributo isOp en para false. Esta torna o dialog (se visível) oculto para o usuário.	<pre>\$dialog.close();</pre>

Property Service

É um serviço declarado através do framework AngularJS que expõe funções públicas para trabalhar com propriedades, encapsulando a complexidade de realizar requisições HTTP para o desenvolvedor. Para maiores informações, leia a documentação de [resources](#).

As funções disponíveis publicamente são:

Nome	Descrição	Utilização
properties	A partir do nome de objeto, busca todas as propriedades deste objeto no back-end.	<pre>\$property.properties({ object: \$object.tlObject })</pre>
range	A partir de um objeto e um atributo, busca a série se existente no arquivo de internacionalização.	<pre>\$property.range({object: 'company', attribute:'enabled'});</pre>

Submit Service

Que expõe funções que mapeiam os métodos genéricos CRUD da arquitetura back-end. Internamente as requisições HTTP são tratadas e processadas, tornando-se transparente para o desenvolvedor.

As funções disponíveis publicamente são:

Nome	Descrição	Utilização
------	-----------	------------

create	Realiza uma chamada HTTP POST para a URL configurada no MessageProvider durante a inicialização da aplicação. Esta configuração mapeia a URL base dos endpoints RESTful genéricos que implementam as operações CRUD. O objeto enviado como primeiro argumento deve conter o atributo object com valor do tipo String, contendo o nome da entidade que será persistida. O segundo parâmetro é o objeto JSON com os dados a serem persistidos. O terceiro e quarto parâmetro são callbacks opcionais, de sucesso e falha. Após sua execução, este método delega ao back-end a responsabilidade de inserção e recebe como resposta, no caso de sucesso, um objeto contendo o id do objeto gerado pelo back-end ou, caso de falha, os erros gerados pelo back-end na inserção.	<pre><code>\$submitService.create({object: \$object.tlObject}, model, successFnCurried, failureCallback);</code></pre>
update	Realiza uma chamada HTTP PUT para a URL configurada no MessageProvider durante a inicialização da aplicação. Esta configuração mapeia a URL base dos endpoints RESTful genéricos que implementam as operações CRUD. O objeto enviado como primeiro argumento deve conter o atributo object com valor do tipo String, contendo o nome da entidade que será atualizada e um atributo id com o identificador unico da entidade já persistida no banco de dados. O segundo parâmetro é o objeto JSON com os dados a serem atualizados. O terceiro e quarto parâmetro são callbacks opcionais, de sucesso e falha. Após sua execução, este método delega ao back-end a responsabilidade de atualização de um objeto já persistido. Se a operação ocorrer com sucesso e houver um callback registrado, este é executado e não recebe nenhum parametro . Em caso de falha, se houver callback de falha registrado, é executado e recebe os erros gerados pelo back-end.	<pre><code>\$submitService.update({object: \$object.tlObject, id: id}, model, successFnCurried, failureCallback);</code></pre>

delete	Realiza uma chamada HTTP DELETE para a URL configurada no MessageProvider durante a inicialização da aplicação. Esta configuração mapeia a URL base dos endpoints RESTful genéricos que implementam as operações CRUD. O objeto enviado como primeiro argumento deve conter o atributo object com valor do tipo String, contendo o nome da entidade que será excluída e um atributo id com o identificador unico da entidade já persistida no banco de dados. O segundo e terceiro parametro são callbacks opcionais, de sucesso e falha. Após sua execução, este método delega ao back-end a responsabilidade de deleção de um objeto já persistido. Se a operação ocorrer com sucesso e houver um callback registrado, este é executado e não recebe nenhum parametro . Em caso de falha, se houver callback de falha registrado, é executado e recebe os erros gerados pelo back-end.	<pre><code>\$submitService.delete({object: \$object.tlObject, id: id}, successFnCurried, failureCallback);</code></pre>
--------	--	---

retrieve	Realiza uma chamada HTTP GET para a URL configurada no MessageProvider durante a inicialização da aplicação. Esta configuração mapeia a URL base dos endpoints RESTful genéricos que implementam as operações CRUD. O objeto enviado como primeiro argumento deve conter o atributo object com valor do tipo String, contendo o nome da entidade que será persistida. O segundo parâmetro é o objeto JSON com os dados a serem persistidos. O terceiro e quarto parâmetro são callbacks opcionais, de sucesso e falha. Após sua execução, este método delega ao back-end a responsabilidade de busca de um objeto já persistido. Se a operação ocorrer com sucesso e houver um callback registrado, este é executado e recebe o objeto encontrado ou nulo, caso não foi possível encontrar um objeto através do identificador informado. Em caso de falha, se houver callback de falha registrado, é executado e recebe os erros gerados pelo back-end.	<pre>\$submitService.retrieve({ object: objectReference, id: lastSelectedId }, viewSuccessFn, viewUpdateErrorFn);</pre>
----------	---	---

find	Realiza uma chamada HTTP GET para a URL configurada no MessageProvider durante a inicialização da aplicação. Esta configuração mapeia a URL base dos endpoints RESTful genéricos que implementam as operações CRUD. O objeto enviado como primeiro argumento deve conter o atributo object com valor do tipo String, contendo o nome da entidade que será persistida. O segundo parâmetro é o objeto JSON com os dados a serem persistidos. O terceiro e quarto parâmetro são callbacks opcionais, de sucesso e falha. Após sua execução, este método delega ao back-end a responsabilidade de deleção de um objeto já persistido. Se a operação ocorrer com sucesso e houver um callback registrado, este é executado e recebe um array de objetos encontrados, ou um array vazio, caso não foi possível encontrar objetos persistidos. Em caso de falha, se houver callback de falha registrado, é executado e recebe os erros gerados pelo back-end.	<pre><code>\$submitService.find({ object: 'period' });</code></pre>
------	---	---

Todos os métodos executados retornam um `$promise`. Promises são promessas de execução, em algum momento no futuro, da operação desejada (ex.: execução de uma chamada HTTP PUT). Algumas directives conseguem trabalhar com promises e com isso eliminamos a necessidade de callbacks. Para maiores informações, consulte a especificação [Promises/A+](#).



Atenção

Estas funções não realizam nenhuma lógica adicional, apenas encapsulam as operações HTTP para a chamada de endpoints RESTful genéricos. Sendo assim, é responsabilidade da camada de serviços a implementação destas operações e a garantia de sua execução.

Directives Disponibilizadas pela Arquitetura

O framework AngularJS já possui um conjunto de directives implementadas e disponíveis para construir a camada de apresentação. Contudo, requisitos como: internacionalização, estruturação, padronização, temas, reuso, não foram totalmente satisfeitos com as [directives](#) já implementadas. Entretanto, foi possível criar directives customizadas que atendem estes requisitos, graças a extensibilidade do framework. Nesta seção mostraremos as directives disponíveis na arquitetura, como utiliza-las em partials e quais são suas funcionalidades.



Todas as diretivas tem uma namespace como prefixo, como por exemplo ng na API do framework AngularJS. Este prefixo serve para que atributos em elementos HTML definidos pela especificação [HTML Spec 4.01](#) não entrem em conflito com nomes de directives. A fim de seguir as boas práticas e evitar problemas advindos de novas especificações, toda directive criada e disponibilizada pela arquitetura é prefixada com a namespace `tl`.

TL Object

A directive `tl-object` é uma das mais importantes directives criadas na arquitetura. Tem como responsabilidade indicar em qual objeto estamos trabalhando em uma determinada UI e em qual contexto que estamos trabalhando, listagem ou detalhe, além de inicializar a busca de mensagens para a internacionalização. Através de herança, outras directives utilizam-se destas duas características para implementar suas funcionalidades e manipular a UI conforme necessário.

Listagem de atributos disponibilizados pela directive `tl-object`:

Atributos	Parâmetros	Descrição
tl-object	String	Em tempo de execução, toda a directive declarada abaixo da directive TL Object (nós filhos do nó onde há o atributo tl-object e tl-context) utilizará esta String para construir suas URLs de endpoints e executar métodos padronizados que refletirão na entidade de back-end. Ex.: Ao preencher o atributo tl-object com o valor <i>period</i>, se a directive tl-submit estiver presente como um filho do nodo atual, uma operação de create na entidade Period pode ser disparada a partir de uma ação do usuário (normalmente ao clicar em um botão de <i>Adicionar</i>).
tl-context	String	Em tempo de execução, toda a directive declarada abaixo da directive TL Object (nós filhos do nó onde há o atributo tl-object e tl-context) que trabalha com manipulação de elementos da árvore DOM utilizará esta String para construir e/ou determinar a forma de apresentação de conteúdos. Ex.: Ao preencher o atributo tl-context com o valor <i>list</i>, se a directive tl-key estiver presente como um filho do nodo atual, descrições e mensagens de validação não serão incluídas no elemento, caso contrário, todas mensagens e validações irão aparecer em um formato padrão.

Além das funcionalidades e comportamentos acima, a directive tl-object possibilita que outras directives se registrem (através de callbacks) e sejam alertadas quando o back-end retornar as mensagens internacionalizadas. Esta directive faz uso do [MessageProvider](#) para inicializar a busca por mensagens de internacionalização.

Abaixo temos um exemplo prático da utilização desta directive em dois contextos diferentes; tela de listagem e detalhe de períodos.

Listagem de Períodos - TL Object
<pre><div id="content" tl-object="period" tl-context="list"> <!-- ... --> <!-- body --> <!-- ... --> </div></pre>

Neste exemplo temos um *outer div*, ou seja, um div que engloba todos os outros elementos presentes na partial, utilizando a directive tl-object. Quando identificada, o seu construtor é executado e a partir dos valores de atributos em elementos HTML, inicializa-se a busca de mensagens para a internacionalização. Toda e qualquer funcionalidade executada por outras directives nesta partial, incorporarão dois parâmetros na sua execução: objeto de negócio e contexto de visualização.

TL Key

A directive tl-key foi criada para atender o requisito de internacionalização e a padronização de layout para mensagens e validações. Esta directive faz uso da directive tl-object, a fim de obter o contexto da partial e então renderizar estes elementos no formato apropriado e, para registrar-se e obter as mensagens de internacionalização assim que a busca for finalizada pelo [MessageProvider](#).

Listagem de atributos disponibilizados pela directive tl-key:

Atributos	Parâmetros	Descrição
tl-key	String	Este atributo deve especificar um objeto (usualmente mapeado no scope responsável por vincular o controller e a partial) que é utilizado pela directive na construção da consulta enviada ao back-end.

Abaixo temos um exemplo prático da utilização desta directive em dois contextos diferentes, tela de listagem e detalhe de períodos, e a estrutura final destas telas após a execução das directives.

Listagem de Períodos - TL Object
<pre><div id="content" tl-object="period" tl-context="list"> <h2 tl-ref="search.period" tl-key="title"/> <!-- ... --> </div></pre>

Em sua execução, alguns elementos serão acrescentados e/ou modificados baseado no contexto adquirido através da directive tl-object.

TL Ref

A directive `tl-ref` visa facilitar a vida do desenvolvedor. Em conjunto com a directive `tl-key`, é possível criar referencias para chaves evitando a repetição de longos prefixos. Internamente, a directive `tl-ref`, assim como a directive `tl-key`, faz uso da directive `tl-object` a fim de obter o contexto e objeto da partial. Contudo, a `tl-ref` não manipula, de forma alguma, a árvore DOM, trabalhando unicamente em conjunto com o [MessageProvider](#).

Listagem de atributos disponibilizados pela directive `tl-ref`:

Atributos	Parâmetros	Descrição
tl-ref	String	Este atributo deve especificar ao menos uma parte da chave de internacionalização. Com este valor a arquitetura cria referencias que podem eliminar o prefixo. Ex.: Para criar uma referencia para as chaves <code>model.company.name</code> e <code>model.company.abbreviation</code>, o atributo <code>tl-ref</code> deve conter uma String com o valor <code>model.company</code>. Após a execução desta directive, directives em nós filhos podem utilizar apenas a parte não referenciada, como por exemplo: <code>name</code> ou <code>abbreviation</code>.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para tornar o processo de desenvolvimento mais produtivo, evitando repetições de prefixos em chaves de internacionalização.

Referencia de Chaves - TL Ref

```
<div tl-object="company" tl-context="detail" tl-dialog tl-opened="detailDialog.isOpen">

    <!-- ... -->

    <!-- Neste elemento é declarado o tl-ref, informando que todos os filhos deste nó referenciarão model.
company -->
    <div class="dialog-page-body" tl-ref="model.company as model">

        <form class="aui">
            <div class="dialog-panel-body panel-body">

                <div class="field-container">

                    <!-- Este elemento é filho da div onde declaramos o tl-ref, ou
seja, este tl-key agora é mapeado para "model.company.name" eliminando
a necessidade de informar o prefixo model.company -->
                    <input type="text" class="field-medium" tl-key="name" ng-model="model.0x1.name" />

                </div>

            <!-- ... -->

        <!-- ... -->

    </div>
```

Internamente, no momento de execução do construtor da directive `tl-ref`, todas as mensagens presentes no [MessageProvider](#), que estão no formato `raw`, são processadas conforme a referencia informada para a directive e colocadas em cache.

TL Find

A directive `tl-find` disponibiliza busca genérica a partir da interação do usuário com a UI sem a necessidade implementar as chamadas HTTP e event handlers. A partir de um objeto preenchido pelo usuário via data-binding (uso da directive [ng-model](#)), internamente a directive percorre os atributos deste objeto e monta uma consulta padrão. Esta consulta padrão leva em consideração os tipos de dados e é construída através de cláusulas AND (por padrão) e a concatenação dos atributos e valores do objeto.

Listagem de atributos disponibilizados pela directive `tl-find`:

Atributos	Parâmetros	Descrição
tl-find	Object	Este atributo deve especificar um objeto (usualmente mapeado no scope responsável por vincular o controller e a partial) que é utilizado pela directive na construção da consulta enviada ao back-end.
tl-result	Array	Este atributo deve especificar um array que será preenchido automaticamente pela directive se houver sucesso na busca.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para realizar buscas genéricas de qualquer entidade já persistida no back-end.

Referencia de Chaves - TL Ref

```
<!-- partial que possui a estrutura apresentada para o usuário, com campos que estão mapeados (estes campos estão omitidos para simplificar o exemplo) em um objeto de nome 'model' no controller deste partial -->
<div id="content" tl-object="company" tl-context="list">

    <!-- ... -->

    <!-- este botão contem a directive tl-find, na qual recebe o objeto model que contem os valores
preechidos pelo usuário -->
    <!-- também esta mapeado o atributo tl-result, no qual diz que a resposta desta busca deve ser mapeada
para o objeto 'list' presente no scope do controller desta partial -->
    <button class="au-button align-right" tl-ref="common" tl-key="button.search" tl-find="model" tl-
result="list" />

    <!-- ... -->

</div>
```

Quando ocorrer um evento de click neste botão, a directive tl-find irá ler o objeto model, construir a consulta, realizar as chamadas ao back-end e preencher o objeto do tipo **Array** com nome list. Qualquer falha durante neste processo é descartada silenciosamente e a operação é interrompida.

TL Dialog

A directive tl-dialog quando aplicada em um elemento do tipo **block**, faz com que todos os elementos filhos sejam encapsulados em um container maior que é representado visualmente como um **dialog**. Esta directive também fornece um objeto a nível de controller, possibilitando que o desenvolvedor manipule suas propriedades de visibilidade.

Para criar este dialog, a factory **\$dialog** esta disponível através de injeção de dependências e fornece métodos para criar um objeto do tipo **Dialog**. Uma referencia deste objeto deve ser informado em alguns atributos da directive, realizando a ligação entre as interações do controller, dialog e directive.

Listagem de atributos disponibilizados pela directive tl-dialog:

Atributos	Parâmetros	Descrição
tl-dialog	-	Este atributo deve especificar manipula o elemento do tipo block para transforma-lo em uma estrutura HTML de dialog.
tl-opened	Boolean	Este atributo deve especificar um valor do tipo Boolean que é utilizado para controlar a visibilidade do dialog pela directive.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para termos a representação visual de dialogs para o usuário.

Componente e Controller - TL Dialog

```
<!-- elemento div (do tipo block) que sinaliza para a arquitetura que este deve ser um dialog através do
atributo tl-dialog -->
<!-- o atributo tl-opened referencia um objeto do tipo "Dialog" e a variável do tipo Boolean "isOpen" -->
<div tl-object="company" tl-context="detail" tl-dialog tl-opened="detailDialog.isOpen">
```

Durante a sua execução, a directive tl-dialog transforma este elemento em um dialog e controla sua visibilidade através da referencia passada para o atributo tl-opened.

Abaixo temos uma representação visual da directive tl-dialog da tela detalhe de Companhias da aplicação Shield.

Company

Company is a top level element used to organize applications and the respectively users/groups in the hourly access windows. Depending on your permissions, you can add, edit or delete one or more companies. Note that you can only delete a company if there's no applications associated with it.

Name

The name of the company.

May not be null.

Abbreviation

The short name of the company.

May not be null. Size must be between 1 and 5 chars.

Status

The status of the company. If enabled, it is available for usage.

Save

Cancel

TL Submit

A directive `tl-submit`, assim como a directive `tl-find`, executa operações no back-end utilizando os endpoints que expoe os métodos genéricos da arquitetura. Porém, a directive `tl-submit` realiza operações de edição, remoção e adição enquanto a directive `tl-find` executa apenas buscas genéricas.

Listagem de atributos disponibilizados pela directive `tl-dialog`:

Atributos	Parâmetros	Descrição
<code>tl-submit</code>	Object	Este atributo deve especificar objeto que deve ser enviado para o back-end.
<code>tl-opened</code>	Boolean	Este atributo deve especificar um valor do tipo Boolean que é utilizado para controlar a visibilidade do dialog pela directive.

TL Action

A directive `tl-action` habilita a edição, deleção e visualização de dados já persistidos pelo back-end através do componente de dropdown list da Atlassian. Esta directive é utilizada em conjunto com o elemento `<table>` e com a directive `tl-action-model` para possibilitar que outros controlers recebam os dados. Cada linha da tabela renderiza um botão mostrando o menu de ações que podem ser executadas pelo usuário em um determinado objeto.

Listagem de atributos disponibilizados pela directive `tl-action`:

Atributos	Parâmetros	Descrição
<code>tl-action</code>	Array	Recebe um array de objetos, itera esta coleção e modificando a estrutura HTML para incluir as opções para o usuário.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para habilitarmos as ações que podem ser tomadas pelo usuário de forma automática.

Componente e Controller - TL Dialog

```

<table class="au1">

    <!-- ... -->

    <!-- neste exemplo a directive esta recebendo o objeto 0x1 do tipo array, através do atributo tl-action
em conjunto com ng-repeat -->
    <tr ng-repeat="company in list.0x1" tl-action="list.0x1">
        <td>{{company.name}}</td>
        <td>{{company.abbreviation}}</td>
        <td>{{status[company.enabled]}}</td>
    </tr>

    <!-- ... -->

</table>

```

Durante sua execução, a directive `tl-action` altera os elementos `<tr>` para incluir a estrutura do menu e disponibilizar as funcionalidades de edição, visualização e deleção.

Abaixo temos uma representação visual da directive `tl-action` da tela listagem de Companhias da aplicação Shield.



TL Action Model

A directive `tl-action-model` recebe o objeto em que ocorreu a ação do usuário, através da directive `tl-action` e atribui para o `scope` onde ela esta presente. A fim de padronizar as ações de usuário, usualmente dois controllers e duas partials são criadas para representar uma determinada entidade; listagem e detalhe, respectivamente. Conforme esta estrutura, utiliza-se a directive `tl-action` na tela de listagem e a directive `tl-action-model` na tela de detalhe para habilitar a manipulação de dados a partir de ações realizadas na directive `tl-action`.

Listagem de atributos disponibilizados pela directive `tl-dialog`:

Atributos	Parâmetros	Descrição
<code>tl-action-model</code>	Object	Define um objeto em scope na tela de detalhe que recebe o objeto em que o usuário realizou uma ação na tela de listagem.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para habilitarmos as ações que podem ser tomadas pelo usuário de forma automática.

Componente e Controller - TL Dialog

```
<!-- o objeto que sofreu a ação através da directive tl-action será transportado para este controller e atribuído a variável de scope 'model' -->
<div ng-controller="CompanyAddDialogCtrl" tl-action-model="model">
</div>
```

Durante sua execução, as directives `tl-action` e `tl-action-model` trocam mensagens utilizando o sistema de eventos do AngularJS para transportar os objetos que sofreram ações nas telas de listagem.

TL Calendar

A directive `tl-calendar` disponibiliza o [widget](#) de calendário da API AUI da Atlassian. Esta directive funciona como um wrapper para a funcionalidade já implementada, facilitando a integração com o AngularJS e eliminando instruções em Javascript que manipulam a [HTML DOM](#).

Listagem de atributos disponibilizados pela directive `tl-calendar`:

Atributos	Parâmetros	Descrição
<code>tl-calendar</code>	-	Esta directive marca um elemento do tipo <code><input></code> para ser renderizado como calendário. Ações de click e focus abrem uma popup contendo as datas, meses, anos, etc.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para habilitarmos um calendário de fácil usabilidade para o usuário final.

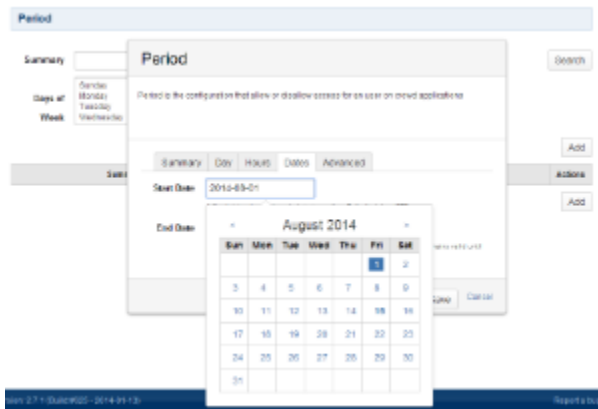
Componente e Controller - TL Dialog

```
<!-- ... -->

    <!-- no input abaixo temos a presença da directive tl-calendar que transforma este input em um calendário -->
    <input tl-calendar ng-model="view.startDate" tl-key="startdate" ng-change="splitStartDate()">
<!-- ... -->
```

Durante sua execução, a directive `tl-calendar` transforma esta directive anexando `<div>`s para construir a estrutura HTML necessária para montar o calendário.

Abaixo temos uma representação visual da directive `tl-calendar` da tela detalhe de Períodos da aplicação Shield.



TL Tab

A directive `tl-tab` disponibiliza o [widget](#) de tab-panels (abas). Dividida em dois atributos, `tl-tabset` e `tl-tab`, a directive `tl-tab` estrutura as abas de forma hierárquica. O atributo `tl-tabset` deve estar presente em um outer element ou nó pai, tendo como nós filhos o número desejado de elementos `<div>` contendo o atributo `tl-tab`.

Listagem de atributos disponibilizados pela directive `tl-calendar`:

Atributos	Parâmetros	Descrição
<code>tl-tabset</code>	-	Marca um elemento do tipo block como o nó pai de todas as abas.
<code>tl-tab</code>	-	Marca um elemento do tipo block como o nó filho, alterando sua renderização para abas e sua visibilidade a partir do atributo <code>tl-active</code> .
<code>tl-active</code>	Boolean	Manipula a visibilidade da aba de acordo com o valor do tipo Boolean obtido através da variável passada para este atributo.

Abaixo temos um exemplo prático de como esta directive pode ser utilizada para habilitarmos um calendário de fácil usabilidade para o usuário final.

Componente e Controller - TL Dialog

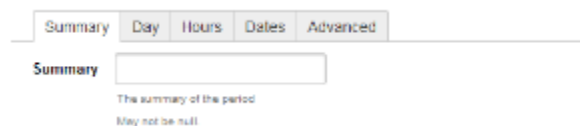
```
<!-- inicializa este elemento como nó pai de todas as abas, controlando sua visibilidade -->
<div tl-tabset>

    <!-- transforma esta div em uma aba, internacionaliza seu título através da directive tl-key e
    determina a visibilidade através do atributo active apontando para a variável "active" nos objetos "view.tabs.
    dates" -->
    <div tl-tab tl-key="detail.period.dates" active="view.tabs.dates.active">

        <!-- ... -->

    </div>
</div>
```

Abaixo temos uma representação visual da directive `tl-tab` da tela detalhe de Períodos da aplicação Shield.



Camada de Serviços

Bem como o objetivo da camada de visualização, a camada de serviços também tem o propósito de trazer facilidades para o desenvolvimento de aplicações, tentando amenizar os problemas ocasionados por um desenvolvimento sem padrões definidos e no modo "go horse". Foi pensando nestes objetivos que surgiu o projeto COMMONS-BUSINESS, que contempla a camada de serviços e a camada de persistência.

A camada de serviços nada mais é do que uma adaptação do padrão de [Restful Objects](#). Ela é responsável por oferecer uma forma de comunicação entre as diferentes partes do plugin através de [URI](#) que são expostos como serviços.

O modelo de URI é baseado nas operações HTTP (POST, PUT, DELETE e GET). E tratando-se do padrão Restful a URI do serviço exposto tem uma estrutura diferente, mantendo, entretanto, sua estrutura base, conforme podemos ver a seguir..

`http://{host}:{port}/{context}/rest/shield/latest/domain/{object}/`

Onde:

Variável	Descrição
{host}	Define onde a aplicação esta rodando
{port}	É a porta do host onde a aplicação esta rodando
{context}	É o Servlet Context da aplicacao. Por exemplo, no crowd normalmente é /crowd
{object}	É o objeto da camada de persistência. Ou seja, é a classe Java que representa para a aplicação a entidade de banco de dados

Operações suportadas

Conforme dito acima, a camada de serviços expõe serviços que podem ser acionados através de operações HTTP, são elas:

Operação	Descrição
DELETE	É a forma utilizada para remover registros identificados através da URI.
GET	É usado para buscar ou ler recursos do banco de dados. A representação desse recurso pode ser expostos em alguns formatos, bem como XML ou JSON.
POST	É utilizado geralmente para a criação de novos recursos.
PUT	É geralmente utilizado para a atualização de recursos

Leia mais: [HTTP Methods](#)

A seguir, as mesmas serão exploradas, dando exemplos utilização através de arquitetura.

GET

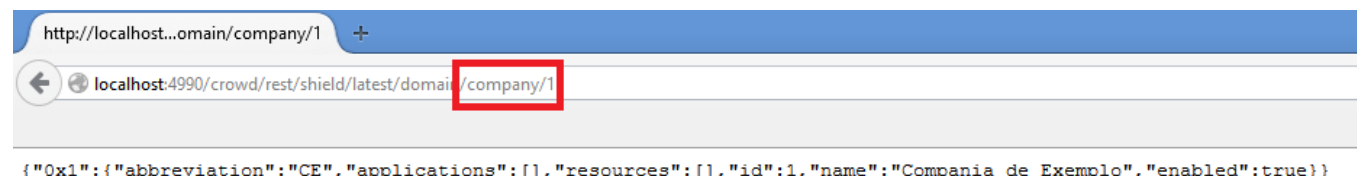
Como já dito, o método GET é utilizado para resgatar recursos de uma origem de dados. No caso da arquitetura criada, essa operação pode ser feita de duas maneiras. A primeira é passando o ID do objeto a ser buscado e a segunda é passando uma condicional para buscar uma lista de um determinado objeto.

Estas duas maneiras serão abordadas a seguir:

Passando ID:

URI: `~/{id}`

É a maneira simplificada de resgatar objetos através do serviço, onde será retornado um registro único caso o mesmo exista. Assim sendo, a forma de utilizar este serviço é através da URI base concatenando ela com o identificador único (ID) do objeto a ser resgatado. A camada de serviço interpretará a operação requisitada e realizará uma pesquisa utilizando o identificador passado retornando o referido objeto, caso ele exista. Abaixo podemos ver um exemplo:



Onde:

Parâmetro	Descrição
company	Este é o objeto representativo do banco de dados. Ou seja, este parâmetro substitui o atributo {object} da URI base da camada de serviço. Neste caso, existe uma classe "Company" que representa uma tabela do banco de dados
1	É o identificador único (ID) da entidade company no banco de dados.

Assim sendo, no exemplo acima foi requisitada uma operação do tipo GET para o serviço, tentando buscar uma "company" com ID "1". O retorno desta consulta, como podemos ver, é uma String com os atributos do classe representativa da entidade de banco de dados Company.

Usando condicional:

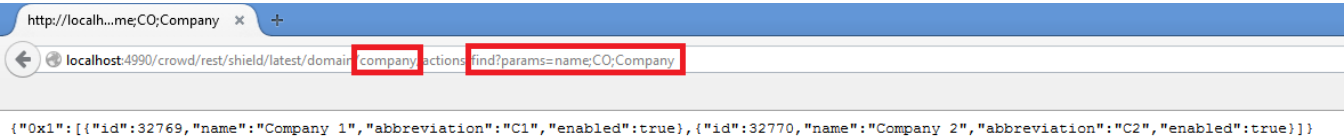
URI: ~/actions/find

É a maneira que a camada de serviço realiza operações de busca um pouco mais complexa. Nela são utilizados operadores pré-determinados pela arquitetura conforme podemos ver na listagem abaixo:

Constante	Representação em SQL	Descrição	Exemplos	URI
EQ	=	É o operador de igualdade. Ou seja, através deste, são feitas comparações entre dois objetos dizendo se são iguais.	1 = 1: Verdadeiro 1 = 2: Falso "teste" = "teste": Verdadeiro "teste" = "teste1": Falso	URI~actions/find?params=enabled;eq>true
LTE	<=	É o operador que diz se um registro é menor ou igual a outro. Pode ser usado somente com números.	1 <= 1: Verdadeiro 1 <= 2: Verdadeiro 2 <= 1: Falso	
GTE	>=	É o operador que diz se um registro é maior ou igual a outro. Pode ser usado somente com números.	1 >= 1: Verdadeiro 1 >= 2: Falso 2 >= 1: Verdadeiro	
LT	<	É o operador que diz se um registro é menor a outro. Pode ser usado somente com números.	1 < 1: Falso 1 < 2: Verdadeiro 2 < 1: Verdadeiro	
GT	>	É o operador que diz se um registro é maior a outro. Pode ser usado somente com números.	1 > 1: Falso 1 > 2: Falso 2 > 1: Verdadeiro	
ST		Não especificado ainda		
EN		Não especificado ainda		
CO	CONTAINS	É o operador utilizado para verificar se um determinado valor esta contido num campo	CONTAINS (nome, "Joao")	URI~actions/find?params=abbreviation;co;C1
NST		Não especificado ainda		
NEN		Não especificado ainda		
NCO		Não especificado ainda		
IN	in	É a condicional que verifica se um determinado valor está em um conjunto de dados.	1 in (1,2,4,6): Verdadeiro 1 in (2,4,6,8): Falso	
IS	is			

NEQ	!=	É o inverso do operador "=". Ou seja, ele verifica se um valor é diferente de outro.	1 != 1: Falso 1 != 2: Verdadeiro "teste" != "teste": Falso "teste" != "teste1": Verdadeiro	
-----	----	--	---	--

Esses parâmetros são enviados juntamente com a requisição HTTP com a finalidade de restringir os resultados na hora de realizar uma busca. Abaixo temos um exemplo de sua utilização:



Onde:

Parâmetro	Descrição
company	Este é o objeto representativo do banco de dados. Ou seja, este parâmetro substitui o atributo {object} da URI base da camada de serviço. Neste caso, existe uma classe "Company" que representa uma tabela do banco de dados
find? params=name; CO;Company	São as condicionais utilizadas para restringir o retorno da busca. Neste caso, o operador utilizado é o "CO" (contains, em SQL) buscando no campo "name" algum registro que contenha "Company". Se convertermos isso em SQL, a instrução seria: Select * from company c where c.name like '%Company%'

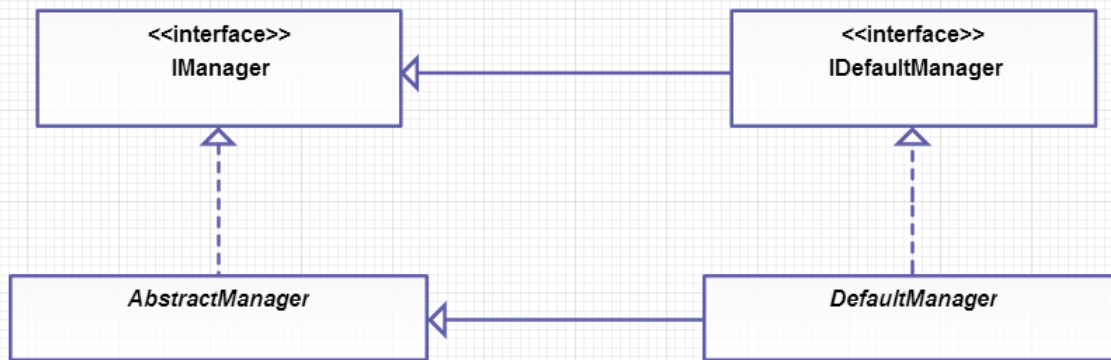
POST

PUT

DELETE

Camada de Negócios

A camada de negócio provida pela arquitetura, é dada através das classes com sufixo Manager (dentro projeto 3layer-commons-buiness). Ela é estruturada da seguinte maneira:



Esta camada fornece métodos CRUD com implementação básica e são expostos dentro da arquitetura para que, caso seja necessário, possam ser implementados de forma específica. Os métodos declarados na interface **IManager** e Implementados na classe **AbstractManager** e são:

```

▼ AbstractManager
  AbstractManager(IBasicPersistence)
  create(T): T ↑IManager
  get(Class<T>, Long): T ↑IManager
  find(Class<T>, Object...): List<T> ↑IManager
  delete(T): void ↑IManager
  update(T): T ↑IManager
  validateCreate(T): T ↑IManager
  validateUpdate(T): T ↑IManager
  validateDelete(T): T ↑IManager
  
```

Especificando Métodos de Negócio da Arquitetura

Caso sejam necessárias criar regras e validações específicas para um determinado fluxo, o mesmo pode ser feito seguindo os seguintes passos:

1. Criar uma interface que estenda **IManager** tipificando-a com a Classe referente a entidade de banco de dados (para fins de persistência de dados)

Interface

```

1 public interface IExemploManager extends IManager<EntidadeBancoDados>{
2 }
  
```

2. Cria uma classe concreta que implemente a interface criada e estenda **AbstractManager** tipificando-a com a Classe referente a entidade de banco de dados (para fins de persistência de dados)

Classe Concreta

```

1 public class ExemploManager extends AbstractManager<EntidadeBancoDados> implements IExemploManager{
2
3     public ExemploManager(IBasicPersistence persistence) {
4         super(persistence);
5     }
6 }

```

3. Sobrescrever os métodos que precisam ser especificados (neste exemplo, especificaremos a validação)

SobrescritaMetodo

```

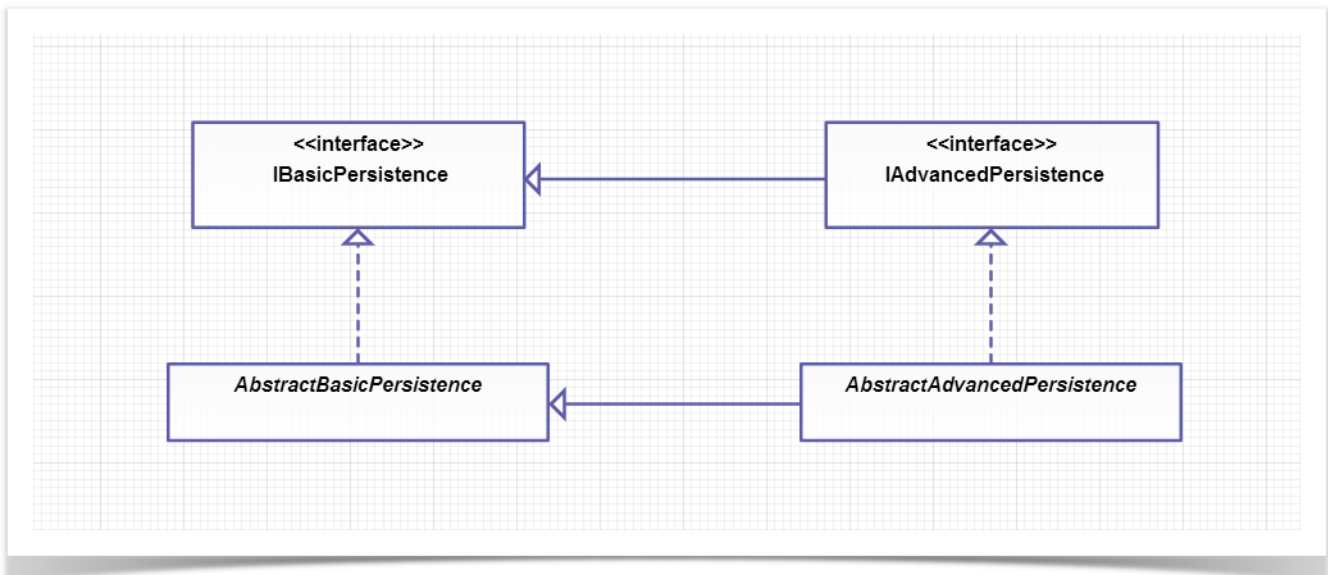
1 @Override
2 protected Period validateCreateImpl(EntidadeBancoDados model) {
3     // Aplicar regras de negócio
4     return super.validateCreateImpl(model);
5 }

```

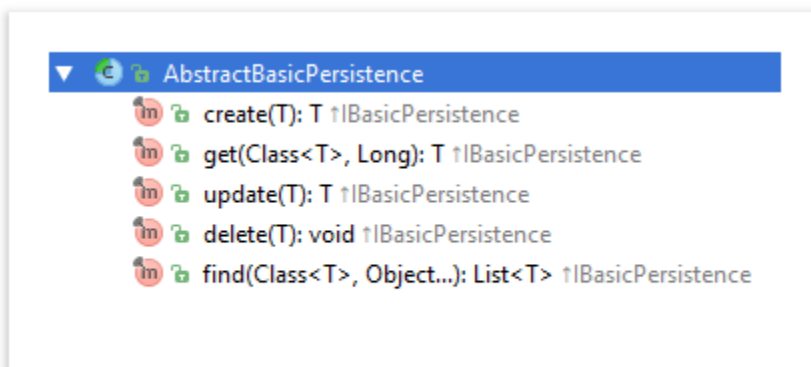
4. Chamar normalmente o método e através do delegate o método específico será acionado

Camada de Persistência

A camada de persistência provida pela arquitetura é dada através das classes com sufixo Persistence (dentro projeto 3layer-commons-business). Ela é estruturada da seguinte maneira:



Neste diagrama podemos destacar a interface IBasicPersistence e a classe AbstractBasicPersistence.



Estes métodos são responsáveis por realizar a persistência na base de dados e da mesma maneira como ocorre com a camada de negócio, a camada de persistência pode ter especificações