MM - Modelo Arquitetural do Framework

MM-60 - O projeto Jira não existe ou você não tem permissão para visualizá-lo.

Nesta pagina

Informações sobre o Modelo Arquitetural da Aplicacao, conhecido como MAS.

CONTEUDO

Secoes Interessantes a Serem Feitas

- Contextualizacao / Proposito da Arquitetura
- Tecnologias Utilizadas
- Camadas da Arquitetura
- Fora de escopo (ou seja, aquilo que a arquitetura nao faz - pelo menos na versao atual)
- Componentes da Solucao (Quais as partes - que existem dentro das camadas - da arquitertura, e o que cada uma faz e como elas se relacionam com outras)
- Estrutura Logica de Dependencias dentre os Componentes / Projetos (maven, IDE) da arquitetura
- Exemplo Pratico (passo a passo, desde um checkout inicial para fazer uma tela - usar como exemplo os tres casos: `Compania`, `Periodo` e `Aplicacao`)
- FAQ
- Mais ideias ?

1. Propósito da Arquitetura

Após analisar o histórico de projetos já desenvolvidos na empresa, foi possível identificar diversos padrões que poderiam ser aplicados a fim de eliminar as divergências detectadas. Estes trabalhos utilizavam uma arquitetura JEE baseada em frameworks de mercado com a proposta de facilitar o processo de desnvolvimento, agilizando as tarefas comuns no dia-a-dia dos desenvolvedores. Infelizmente, por falta de suporte, documentação e treinamento, erros comuns, muitas vezes básicos, prejudicaram as entregas. Consequentemente, a base de código-fonte que deveria ser sólida e limpa, tornou-se desestruturada e de difícil manutenção.

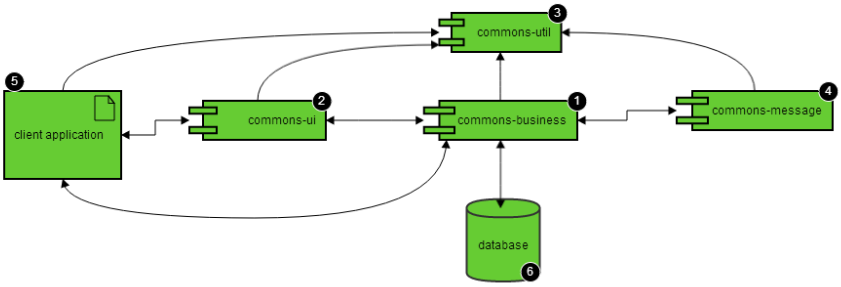
Visando eliminar os problemas previamente citados, uma nova arquitetura, também composta por frameworks de mercado, foi proposta. Com os objetivos traçados, inicia-se a construção de uma nova arquitetura, divididas em três camadas, utilizando o padrão arquitetural MVC para compor as camadas de apresentação, controle e modelo. Estas camadas, por sua vez, também são compostas por padrões de arquitetura e adaptando-os a sua necessidade.

A camada de apresentação utiliza o framework AngularJS, o mais novo lançamento do time de desenvolvedores do Google. Diferentemente de outros frameworks JavaScript, ele adota uma abordagem mais ligada à sintaxe HTML, funcionando como uma espécie de extensão da linguagem. Componentes padrões, como por exemplo laços de repetição (ng-repeat), já estão inclusos no framework. Entretanto, novos componentes customizados foram desenvolvidos para auxiliar nossos desenvolvedores. Estes componentes são classificados no AngularJS como directives, filters, providers.

A camada de serviços faz uso da especificação JSR311 e sua implementação de referência Jersey. Em conjunto, ambos buscam simplificar o desenvolvimento de serviços REST e oferecer uma forma padrão de implementação desta linha de serviços. O Jersey implementa os recursos definidos pela JSR e acrescenta algumas funcionalidades, como o WADL e uma API para clientes REST. Novamente, a arquitetura provê métodos auxiliares, nos quais implementam operações CRUD através de endpoints RESTful.

Por fim, temos a camada de negócios e as diferentes implementações de frameworks que disponibilizam o mapeamento de objeto-relacional, são eles: Hibernate, AO, TopLink. Nesta camada foi possível criar abstrações para que seja transparente as operações nativas destes frameworks, isto é, o desenvolvedor não preocupa-se com detalhes de qual framework está sendo utilizado pela aplicação. Contudo, estas não são as únicas funcionalidades da camada. Validações de modelo e regras de negócio também estão contidas neste terceiro nível.

Assim, após a apresentação formal do propósito da arquitetura, concluimos esta seção com uma ilustração da comunicação e dependência entre as camadas. Nesta ilustração é possível perceber a interação entre as diferentes responsabilidades e componentes.



e

Onde:

Item	Nome	Descrição
1	COMMONS-BUSINESS	É a camada responsável por disponibilizar os métodos padrões de negócio e de serviço.
2	COMMONS-UI	É a camada da arquitetura que provê soluções para as interfaces de usuário.
3	COMMONS-UTIL	
4	COMMONS-MESSAGE	
5	PRODUCTS	São os produtos desenvolvidos pela empresa que consomem a arquitetura proposta. Como exemplo, podemos citar Shield, Catalog Service.
6	DATABASE	

2. Tecnologias

Fabricante	Tecnologia	Versao	Usada para
Oracle	Java	1.6	Linguagem de programação
Google	AngularJS	1.2.14	Framework MVVM para data-binding e estruturação da camada View/Controller
Google	AngularJS Cookies	1.2.14	Módulo de Cookies para o AngularJS utilizado pelo framework
Google	AngularJS Resouce	1.2.14	Módulo de Resources para o AngularJS utilizado pelo framework
Google	AngularJS Route	1.2.14	Módulo de Route para o AngularJS utilizado pelo framework
Comunidade	jQuery	1.8.3	Biblioteca JS para manipulação de elementos DOM, EventHanlders, AJAX, etc.
Atlassian	AJS	5.2	Biblioteca JS de componentes da Atlassian
Atlassian	AUI	5.2	Conjunto de folhas de estilo (CSS) para padronização de UI
Google	GSON	2.2.4	Mapeamento Java/JSON
Codehaus	Jackson	2.1.2	Mapeamento Java/JSON (não utilizado, vem junto com o pacote de depêndencias da Atlassian)
Codehaus	MVEL	2.2.0. Final	Mapeamento de POJO/DTO com o projeto Murdoc
Oracle	Java Bean Validation	1.1.0. Final	Validação de POJOs na persistência e métodos facilitadores no framework
JBoss	Hibernate Validator	5.1.0. Final	Implementação da especificação (necessário para utilizar o Java Bean Validation)
Oracle	Jersey	1.0.2	Utilizado para mapear endpoints RESTful
Oracle	JAXB	2.1	Mapeamento Java/XML (não utilizado, vem junto com o pacote de depêndencias da Atlassian)
Oracle	Servelet API	2.4	API de servlet Java utilizada pelo servidor de aplicação
Pivotal Software	Spring Framework	2.5.5	Framework utilizado internamente pelas aplicações Atlassian para facilitar a injeção de dependências, OSGI, controle transacional etc

Google	Guava	4.8	Biblioteca de utilidades do Google
JUnit	JUnit	4.8	Framework para testes unitários
JBoss	Hibernate	4.2.4. Final	Framework de persistência

3. 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.

3.1. Camada de Visualização

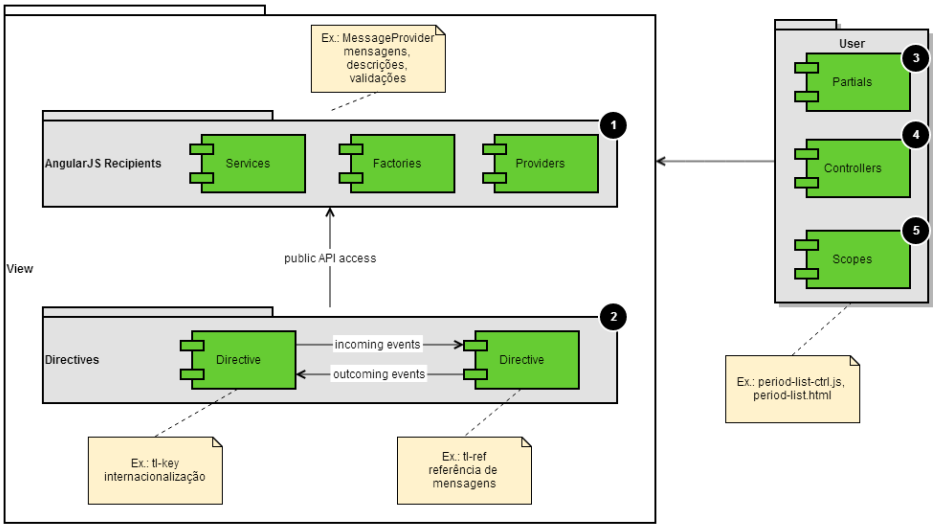
Conforme citado na seção de contextualização, a arquitetura proposta visa eliminar os problemas identificados em diversos projetos já executados, onde foi possível identificar a falta de padronização para realizar tarefas comuns no processo de desenvolvimento. Assim, temos como objetivo principal desta 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 funcionlidades 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 breviamente 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 expoem 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 *ng-include*. 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.

3.2. 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.

3.2.1. 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 quaisquers outros componentes do AngularJS. Abaixo temos um exemplo da utização 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.commons.ui.util
angular.module('treelayer.commons.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, expoem 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.

3.2.2. 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 quaisquers 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 esta 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 constroi 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` .

3.2.3. 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 uteis 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.commons.ui').provider('$message', function () {

        // ...
        this.baseUrl = '';

        // ...

        this.$get = ['$q', '$http', function ($q, $http) {
        return this.$message($q, $http);
    }];

        // ...
});
```

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

A injeção de dependências de providers mudam 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, mantem-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.

3.3. 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 exemlo 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`.

3.4. 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 partials period-list.html e period-detail.html.

3.5. 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 partials) 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 partial period-list.html e do controller `PeriodCtrl`.

3.6. 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 observer 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" 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'}),

                        // ...

                };

        // ...

}]);
```

No exemplo acima, acessamos o scope do controller `PeriodCtrl` através de injeção de dependencias no construtor do controller (variável declarada como `$scope`). Valores são expostos para partials e directives através do operator `.` (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 delcaração `view.status`.

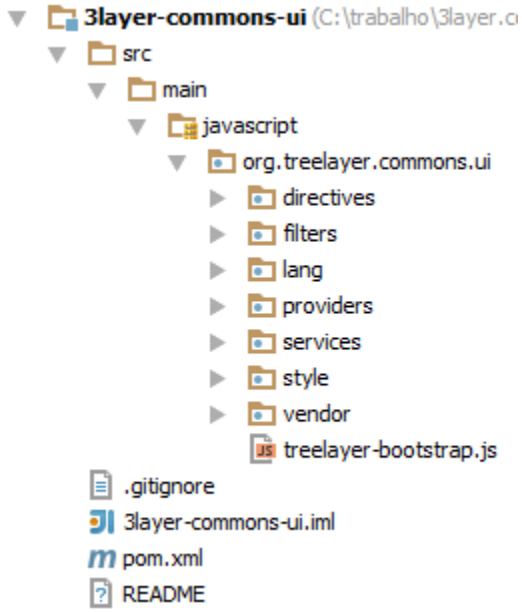
Código Atualizado

Acesse o código-fonte do partial period-list.html e do controller `PeriodCtrl`.

Não deixe de ler a documentação oficial, funcionalidades como: prototypal inheretance, listeners, observers, data-biding, etc, são importantes no desenvolvimento da camada de apresentação.

3.7. 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 dependencias 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 dependencias. Este arquivo existe para que quando necessário, projetos especifiquem como dependência apenas este módulo e todos os componentes estarão automaticamente associados ao projeto.

3.8. 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.

3.8.1. 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`	Fornece uma forma de se registrar a este provider e, quando o back-end retornar, o provider irá alertar os componentes registrados.	`$message.whenMessageLoads(self.cacheKey, function(cacheKey) { self.executeReferenceCallbacks(cacheKey); self.executeKeyCallbacks(cacheKey); });`
`getRawMessages`	Retorna um objeto contendo todas as mensagens, validações e descrições. Este objeto não sofreu nenhuma modificação por directives, esta no formato *raw*.	`curriedProcessMessages($message.getRawMessages($super.cacheKey));`
`getValidations`	Retorna um objeto contendo todas as validações.	`$message.getValidations();`
`createReference`	Usado principalmente pela directive tl-ref, 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.	`/** * 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)`
`findMessagesByKey`	Busca uma determinada mensagem pela chave contida no arquivo property.	`$message.findMessagesByKey('common.button.view');`

3.8.2. Busca de Mensagens

A partir de uma chamada da principal directive, ***tl-object***, é disparada uma requisitção 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.

3.8.3. 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.

3.8.4. 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 *rav* . 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*.

3.9. 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.

3.9.1. 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.new();`

3.9.2. 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.	`tl-operation="detailDialog.operation"`
`isOpen`	Variável de controle utilizada para determinar se o `Dialog` está visível.	`<div tl-object="period" tl-context="detail" tl-dialog tl-opened="$parent.view.detailDialog.isOpen">`

As funções disponíveis publicamente são:

Nome	Descrição	Utilização
`new`	Inicializa um objeto do tipo `Dialog` e retorna.	`$scope.detailDialog = $dialog.new();`

`open`	Altera o valor da atributo `isOpen` 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: *create*, *update*, *delete* e *view*.	`<button class="aui-button align-right" ng-click="detailDialog.open('create')" tl-key="button.add" tl-save="model" />`
`close`	Altera o valor da atributo `isOpen` para `false`. Esta torna o dialog (se visível) oculto para o usuário.	`$dialog.close();`

3.9.3. 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.	`$property.properties({ object: $object.tlObject })`
`range`	A partir de um objeto e um atributo, busca a série se existente no arquivo de internacionalização.	`$property.range( {object: 'company', attribute:'enabled'} );`

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

`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 excluida e um atributo id com o identificador unico da entidade já persistida no banco de dados. O segundo é 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.	`$submitService.delete({object: $object.tlObject, id: id}, successFnCurried, failureCallback);`

`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 é quarto parametro 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.	`$submitService.retrieve({ object: objectReference, id: lastSelectedId }, viewSuccessFn, viewUpdateErrorFn);`

`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 é quarto 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 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.	`$submitService.find({ object: 'period' });`

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 exeução.

3.10. 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**.

3.10.1. 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 característica 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 *period*, 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 *Adicionar*).
`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 *list*, se a directive tl-key estiver presente como um filho do nodo atual, descrições e mensagens de validação não serão inclusas no elemento, caso contrário, todas mensagens e validações irão aparecer em um formato padrão.

Além das funcionalidades e comportamentos acima, a directive tl-object possibilida 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

```
<div id="content" tl-object="period" tl-context="list">

        <!-- ... -->

        <!-- body -->

        <!-- ... -->

</div>
```

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, incializa-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.

3.10.2. TL Key

A directive `tl-key` foi criada para atender o requisito de internacionalização e a padronização de leiaute 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

```
<div id="content" tl-object="period" tl-context="list">

        <h2 tl-ref="search.period" tl-key="title"/>


        <!-- ... -->

</div>
```

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

3.10.3. 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 `model.company.name` e `model.company.abbreviation`, o atributo `tl-ref` deve conter uma `String` com o valor `model.company`. Após a execução desta directive, directives em nós filhos podem utilizar apenas a parte não referenciada, como por exemplo: `name` ou `abbreviation`.

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.

3.10.4. 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 simplicar 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="aui-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.

3.10.5. 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.

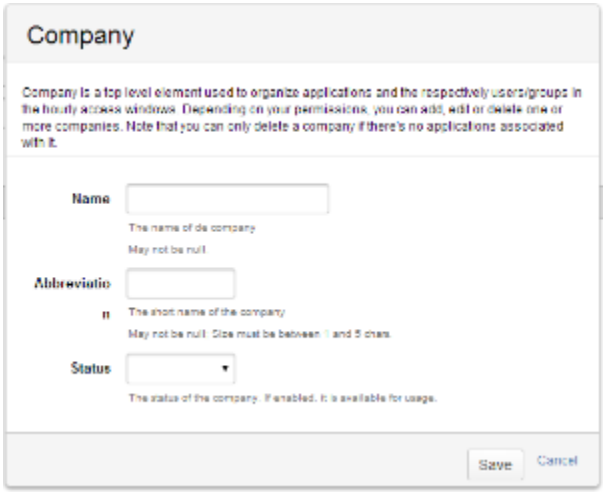
Abaixo temos um exemplo prático de como esta directive pode ser utilizada para termos a representação visual de dialogs para o usuário.

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.



3.10.6. 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
`tl-submit`	Object	Este atributo deve especificar objeto que deve ser enviado para o back-end.
`tl-opened`	Boolean	Este atributo deve especificar um valor do tipo `Boolean` que é utilizado para controlar a visibilidade do dialog pela directive.

3.10.7. 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
`tl-action`	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="aui">

        <!-- ... -->

        <!-- 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.

3.10.8. 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
`tl-action-model`	Object	Define um objeto em scopo 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 scopoe '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.

3.10.9. 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
`tl-calendar`	-	Esta directive marca um elemento do tipo `<input>` 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.

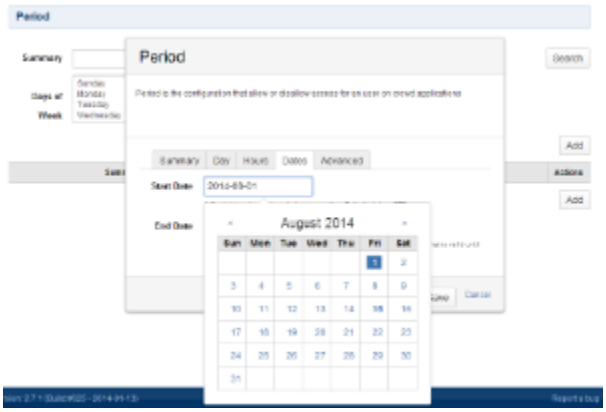
Componente e Controller - TL Dialog

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

        <!-- 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.



3.10.10. 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 contendo o atributo `tl-tab`.

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

Atributos	Parâmetros	Descrição
`tl-tabset`	-	Marca um elemento do tipo block como o nó pai de todas as abas.
`tl-tab`	-	Marca um elemento do tipo block como o nó filho, alterando sua renderização para abas e sua visibilidade a partir do atributo `tl-active`.
`tl-active`	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

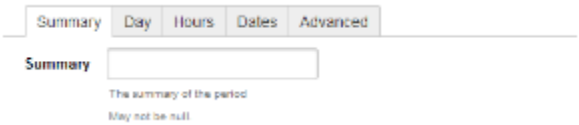
```
<!-- inicializa este elemento como nó pai de todas as abas, controlando sua visibilidade -->
<div tl-tabset>

        <!-- transforma esta div em uma aba, internacionaliza seu título através da directive tl-key e
determina a visiblidade 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>
```

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



3.11. 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

3.11.1. 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.

3.11.1.1. GET

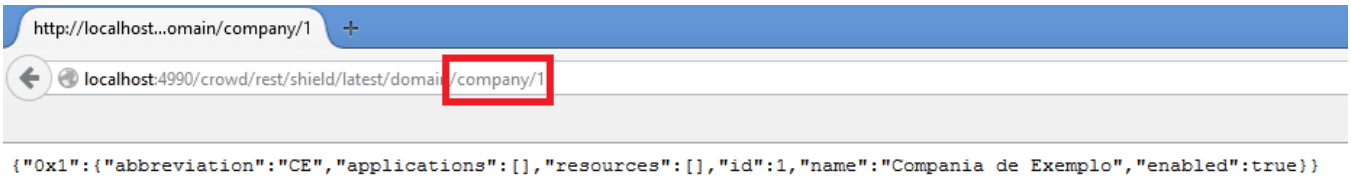
Como já dito, o método GET é utilizado para resgatar recursos de uma origem de dados. No caso da arquitetura criada, essa operação pode ser feita de duas maneiras. A primeira é passando o ID do objeto a ser buscado e a segunda é passando uma condicional para buscar uma lista de um determinado objeto.

Estas duas maneiras serão abordadas a seguir:

3.11.1.1.1. 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.

3.11.1.1.2. Usando condicional:

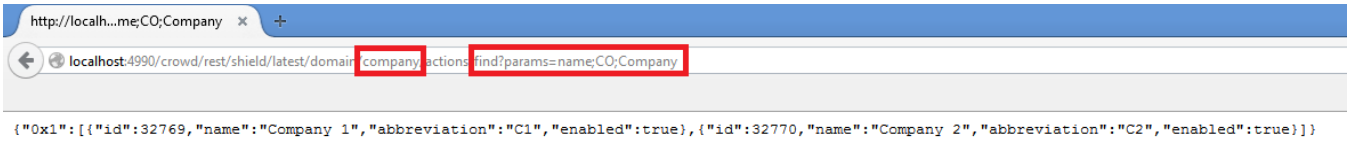
URI: *~/actions/find*

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%'`

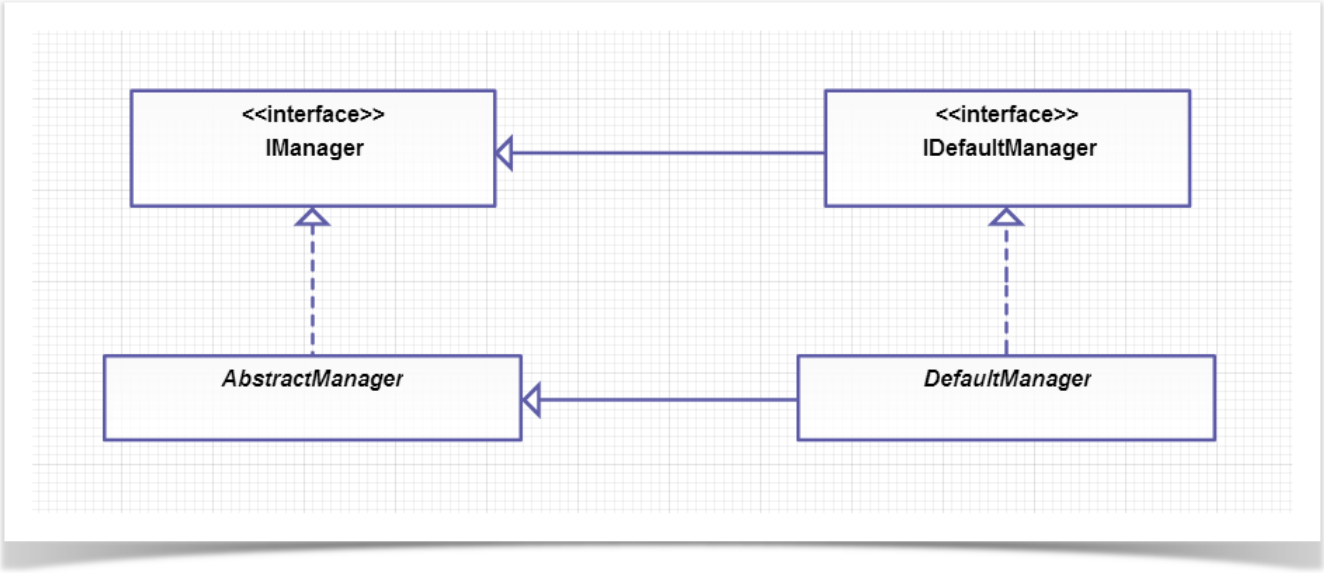
3.11.1.2. POST

3.11.1.3. PUT

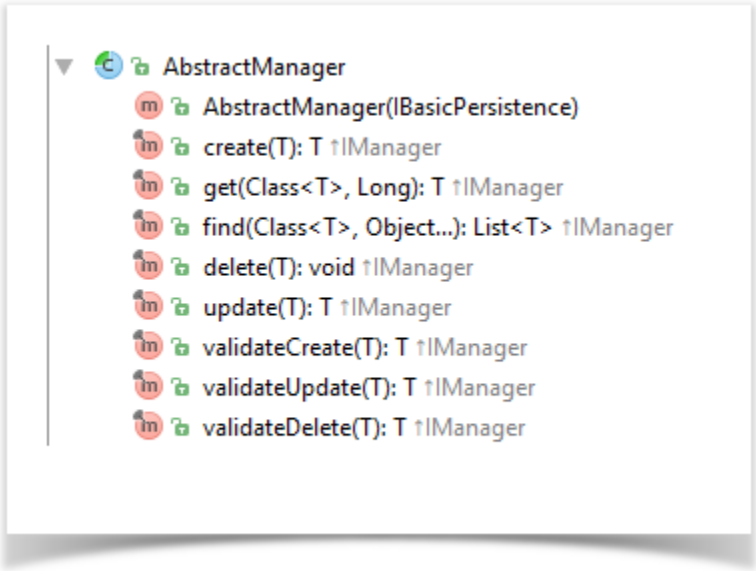
3.11.1.4. DELETE

3.12. 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:



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



3.12.1. Especificando Métodos de Negócio da Arquitetura

Caso sejam necessárias criar regras e validações especificas 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

```
 public class ExemploManager extends AbstractManager<EntidadeBancoDados> implements IExemploManager{

    public ExemploManager(IBasicPersistence persistence) {
        super(persistence);
    }
}
```

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

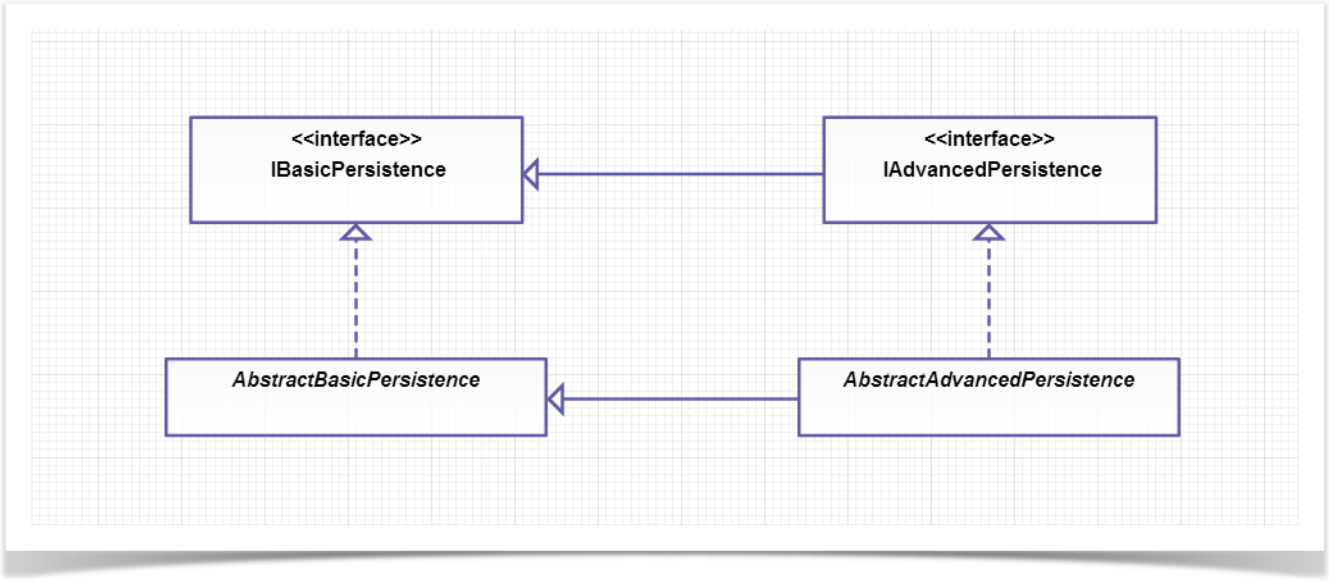
   SobrescritaMetodo

```
@Override
protected Period validateCreateImpl(EntidadeBancoDados model) {
    // Aplicar regras de negócio
    return super.validateCreateImpl(model);
}
```

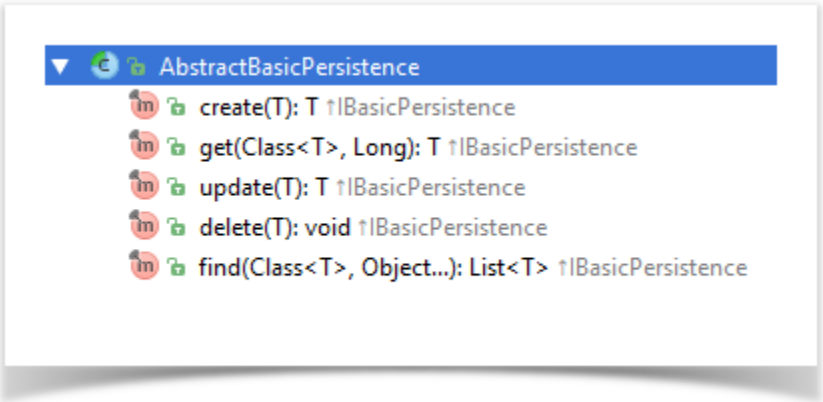
4. Chamar normalmente o método e através do delegate o método especifico será acionado

3.13. Camada de Persistência

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



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



Este 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

4. Escopo

Esta seção aborda capacidade e limitações da atual versão da arquitetura e utiliza-se da aplicação modelo (Shield) como exemplo.

4.1.1. Capacidades

- **Injeção de elementos HTML através de diretivas customizadas:** a diretiva ***tl-key*** fornecida pela arquitetura tem como responsabilidade realizar as injeções de conteúdo e elementos, mutando-se conforme as tags em que é declarada. Esta diretiva esta intrinsecamente ligada as diretivas: ***tl-ref***, ***tl-object*** e ***tl-context***. Todas, em colaboração fornecem aos usuários da arquitetura: internacionalização e injeção de elementos HTML. Abaixo temos um exemplo ilustrativo de seu funcionamento.

Código HTML

```
<div class="inline-field-container">
        <input type="text" class="field-short" tl-key="name" ng-model="model.name" />
</div>
```

Código HTML gerado para o contexto de detalhe

```
<div class="inline-field-container ng-scope">
        <label>Name</label>
        <input type="text" class="field-medium ng-pristine ng-valid" tl-key="users.name" ng-model="model.0x1.
name" id="model.window.users.name.1" />
        <div class="description">User display name.</div>
        <div class="description" name="validation.name">May not be null.</div>
</div>
```

Código HTML gerado para o contexto de lista

```
<div class="inline-field-container ng-scope">
        <label>Name</label>
        <input type="text" class="field-medium ng-pristine ng-valid" tl-key="users.name" ng-model="model.0x1.
name" id="model.window.users.name.1" />
</div>
```

- **Internacionalização:** todo e qualquer elemento HTML injetado pela diretiva ***tl-key*** que contenha texto é automaticamente internacionalizado pela arquitetura utilizando os arquivos de mensagens (arquivos com extensão .properties)
- **Métodos genéricos para operações CRUD simples:** todas as entidades através da arquitetura já possuem endpoints RESTful com métodos de operações CRUD disponibilizados pela arquitetura
- **Data-binding na camada de View:** através da diretiva ***ng-model***, todo e qualquer elemento do tipo ***input*** que dispara eventos, atualiza objetos no controller Javascript automaticamente
- **Injeção de dependências:** aplicações podem disponibilizar serviços através da injeção de dependências pelo construtor da classe, tendo como único requisito que esta classe implemente uma interface que declare seus métodos públicos. Logo abaixo temos um exemplo desta implementação através de injeção de dependência utilizando construtores.

```
public interface IInjectable {
        void foo();
}

public class Injectable extends IInjectable {
        public void foo() {
                System.out.println("foo");
        }
}

public class Injected {
        public Injected(IInjectable injectable) {
                injectable.foo();
        }
}
```

- **Mapeamento objeto relacional:** através de frameworks como Hibernate, TopLink ou AO é possível mapear objetos no Java que reflitam as tabelas da base de dados.
- **Controle Transacional (limitado):** cada chamada HTTP é processada por um filtro de servlet que inicia uma transação, encaminha as operações e finaliza a transação se tudo ocorreu com sucesso. Se erros ocorrerem durante este processo toda a chamada é invalidada. Esta funcionalidade não possui suporte para transações aninhadas.
- **Padronização de elementos UI através de classes CSS:** o pacote AUI da Atlassian provê diversas classes para padronizar o laiaute de telas. Classes da arquitetura foram desenvolvidas para trabalhar em conjunto com este pacote e prover uma UI padronizada.
- **Widgets (calendário, modal panel, tab panel):** Diretivas ***tl-dialog***, ***tl-tabset***, ***tl-tab***, ***tl-calendar*** fornecem widgets completos para o desenvolvimento de telas mais complexas e podem ser facilmente incorporadas a telas já existentes.
- **Serialização/Deserialização genérica (podendo ser customizada pelo desenvolvedor):** Todas as chamadas de operações CRUD são tratadas inicialmente pela arquitetura e esta é responsável pela serialização/deserialização das chamadas. Se houver uma sobrecarga destes métodos genéricos, os argumentos recebidos por estes métodos já são serializados e deserializados após o retorno, isto facilita a vida do desenvolvedor ocultando detalhes de serialização/deserialização.
- **Validação de POJOS de forma genérica:** toda chamada para métodos genéricos que provém operações CRUD são validadas automaticamente pela arquitetura. Esta validação pode ser visualizada no snippet do código da arquitetura abaixo.

```
/**
     * Given an object of type T, apply default object validation backed by JSR311 API. If T is a valid object,
then return it,
     * otherwise throws {@link org.treelayer.commons.business.exception.BusinessException} containing a map of
constraint violations
     * as follows:
     *
     * <p>
     * {@code
     * { "exampleEntity.name" -> "javax.validation.NotNull.message",  "exampleEntity.id" -> "javax.validation.
NotNull.message" }
     * }
     * </p>
     *
     * @param model object of type T that will be validated
     *
     * @return validated object
     *
     * @throws org.treelayer.commons.business.exception.BusinessException if object is not valid
     */
    private T applyDefaultValidation(T model) throws BusinessException {
        Map<String, String> constraintViolations = validatorUtil.applyValidation(model);
        if (constraintViolations.size() > 0) {
            throw new BusinessException(HttpURLConnection.HTTP_BAD_REQUEST, constraintViolations);
        } else {
            return model;
        }
    }
```

- **Tratamento de erros de forma transparente:** exceções de negócio são lançadas utilizando a exceção customizada pela arquitetura BusinessException. Esta exceção é tratada pela camada mais externa e transforma as mensagens de erro em mensagens na tela do usuário, já internacionalizadas.

- **Persistência transparente, idependente do framework de persistência utilizado (Hibernate, TopLink, AO):** Todos os managers da arquitetura utilizam uma sólida base de classes que encapsula o framework de persistência utilizado. Com estas classes é possível abstrair as complexidades individuais de frameworks e oferecer assinaturas padrões para o desenvolvedor, não se importando com o framework utilizando.

4.1.2. Limitações

- **Não suporta pilha de modal:** temos na tela de Aplicações um relacionamento ManyToOne com Companhias. A tela de detalhe de aplicação constitui uma modal contendo todos os campos de aplicação e um combobox mostrando as companhias disponíveis. Com a atual arquitetura não é possível editar estas companhias a partir de uma aplicação. Para realizar esta operação seria necessário sobrepor a modal corrente de detalhe com uma modal de datalhe de companhia, controlando o fluxo de operações. A solução é abrir uma nova aba do navegador, realizar a operação e então continuar trabalhando com a aplicação em edição.

- **Não é possível redirecionar o usuário para uma tela de detalhe de outro objeto:** novamente tomando como exemplo a tela de Aplicações e Companhias, ao tentar editar uma companhia vinculada a aplicação corrent, devido a limitação de pilhas de modal, o usuário poderia ser redirecionado para a tela de detalhe da companhia. Esta operação não é suportada pois não é possível alterar as telas e abrir a modal de edição de companhia. A solução é abrir uma nova aba do navegador, realizar a operação e então continuar trabalhando com a aplicação em edição.

- **Ainda não há implementação para lançar mensagens genéricas ou de negócio na tela de lista:** nas telas de detalhe, no topo da modal, logo após a divisão de textos informativo é apresentado uma lista de erros de negócio (se houver erros, caso contrário nada é apresentado). Estes erros são comumente lançados como exceções do tipo BusinessException e tratados pela arquitetura. Esta implementação esta disponível apenas para telas de detalhe, a tela de listas ainda não possui tal funcionalidade.

- **Série de valores para elementos do tipo combobox, multipicker etc, são informadas no arquivo de internacionalização, desta forma perdendo a possibilidade de oferecer configurações para o usuário:** conforme o exemplo de código abaixo, séries são informadas diretamente no arquivo de propriedades que contem as mensagens de internacionalização. O formato {value | label} deve ser seguido para representar propriedades que envolvem séries. Como os valores são informados neste arquivo, não é possível implementar séries configuráveis pelo usuário final.

```
model.period.rule.values.1 = r|Revoked
model.period.rule.values.2 = g|Granted
```

- **Internacionalização é baseada no idioma do navegador e não em um perfil de usuário configurado:** a arquitetura baseia-se no atributo Accept-Language do protocolo HTTP 1.1 para recuperar as configurações de idioma. Conforme podemos ver na imagem abaixo, os idiomas configurados no navegador são passados neste atributo, em uma formatação que envolve separar os idiomas por vírgula e outros valores padrões (facilitando a interpretação por um regex). Se o usuário deseja mudar o idioma, muda-se o idioma no navegador e atualiza-se a página.

5. Ferramentas

Ferramenta	Versao	Utilizadores (perfil na equipe)	Usada para
Intellij IDEA (Community Edition)	13.0.1	Desenvolvedores	Ferramenta do dia-a-dia utilizada pelos desenvolvedores para trabalhar com linguagens voltadas para o back-end (Java)

Git Integration (Intellij Plugin)	2.4	Desenvolvedores	Plugin da IDE Intellij IDEA para trabalhar com o repositório de artefatos
WebStorm	7.0.3	Desenvolvedores	Ferramenta do dia-a-dia utilizada pelos desenvolvedores para trabalhar com linguagens voltadas para o front-end (Javascript)
HipChat	2.2.1163	Todos	Ferramenta de comunicação que possibilita a troca de mensagens por texto, vídeo e envio de anexos
SourceTree	1.5.1.0	Desenvolvedores	Ferramenta para trabalhar com o repositório (Git) e realizar operações do dia-a-dia (pull, push, rebase, fetch)
Stash	3.1.0	Desenvolvedores, Gerentes de Configuração	Ferramenta na nuvem da 3layer que disponibiliza code review, code browser, pull requests etc
Jira	6.2.5	Todos	Ferramenta na nuvem da 3layer que disponibiliza controle de tarefas, gerencimento de projetos, dashboards, registro de horas etc
Confluence	5.5	Todos	Ferramenta na nuvem da 3layer que disponibiliza wiki colaborativa entre os participantes do projeto
Apache Tomcat	6	Desenvolvedores, Gerentes de Configuração	Servidor de aplicação
Chrome	36.0.1985.125 m	Todos	Navegador padrão na validação da aplicação modelo (Shield)
Internet Explorer	9	Todos	Navegador para utilizar na homologação
Apache Maven	2.1.0	Desenvolvedores	Gerência de projetos Java e controle de dependências
Atlassian SDK	4.2.10	Desenvolvedores	Kit de desenvolvimento da Atlassian que permite aos desenvolvedores a construção de plugins, deploy e configuração
Git	1.9.0	Todos	Repositório de artefatos
Balsamiq Mockups	Next.558 - 03/14 /2011 12:18	Analistas, Projetistas, Arquitetos	Criação de mockups
Enterprise Architect	10.0.1009	Analistas, Projetistas, Arquitetos	Criação de diagramas e documentação

Camadas da Aplicacao

Interface do Usuario

Consulte o projeto grafico para um detalhamento dos componentes internos, estrutura, aparecia e comportamento dessa camanda.

Camada de Servicos

Todos os servicos do sistema estao implementados na forma de Enteprise Java Beans, padrao 3.1, que em suma:

- Nao utilizam interface para descricao dos servicos, apenas classe de implementacao diretamente (visando minimizar tempo de programacao)
- Todos os EJB sao locais
- Todos os EJB sao sem estado (@Stateless)
- Todos os EJB sao nomeados (@Named)

Persistencia e Banco de Dados

O Sistema EMapping utiliza a modelagem Orientada a Objetos para suas entidades de negocio e dados. Os dados persistentes sao mapeados para banco de dados relacional utilizando o padrao JPA (JSR-317), com geracao automatizada das estruturas na base de dados atraves da implementacao Hibernate.

A responsabilidade das estruturas OO do sistema ficam a cargo do projetista do sistema, conforme necessidades de negocio elencadas pelo Product Owner e direcionamentos dos arquitetos.

O banco de dados padrao do sistema eh o Oracle, mas devido o uso de JPA, Postgres tambem deve ser suportado.

Todo e qualquer acesso a base de dados deve ser feito unica e exclusivamente atraves do objeto `EntityManager` que esta encapsulado nos EJBs do sistema. Excecoes a essa regra competem apenas ao arquiteto do sistema.

Abaixo, um exemplo de uso da camada de persistencia

```
@Stateless
public class ReferentialDomainService extends AbstractCoreService<AbstractNamedCoreModel> {
  public void persist(ReferentialField field) {
    ReferentialField fieldToPersist = field;
    if (isUsed(fieldToPersist)) {
      fieldToPersist = new ReferentialField(field);
    }
    super.persist(fieldToPersist); //persist eh um metodo implicito e disponivel para os EJBs de servico como
este
  }
}
```

Servicos Externos

Os servicos externos que acessam o EMapping poderao fazer isso atraves do padrao REST, podendo realizar leitura e escrita de informacoes desde que autenticados e conforme o nivel de permissao atribuido.

Os servidos REST disponiveis no EMapping sao implementados unica e exclusivamenete pelos EJB de fachada.

O formato padrao de dados REST no EMapping eh, primeiro XML, e opcionalmente JSON.

Detalhes e exemplos sobre essa parte serao disponibilizados quando necessario.

Funcionamento

Os quatro vertices (Logico, Fisico, Processamento e Implantacao) da arquietetura sao dados pelo Diagrama 4+1 abaixo:

Logico

Fisico

Processamento

O Diagrama de Atividades mostra o ciclo completo de processamento de negocio de um acao de usuario no sistema:

Mockup image file mockup_diagramaDeSequencia.png (version 5) not found. Did someone delete it?

Onde:

- Cores:
  - **Azul**: Em suma eh o navegador web, que executa no computador cliente. Captura as acoes do usuario e converte em requisicoes http /s. Tambem captura os retornos do sistema e exibe-os em uma interface html rica (web 2.0).
  - **Verde**: As actions sao componentes da aplicacao escritos em codigo java. Recebem as requisicoes do browser e realizam validacoes simples (que nao exigem acesso ao sistema). Quando uma acao necessita o processamento de uma regra de negocio, ela aciona um EJB de Fachada.
  - **Laranja**: Sao EJBs. Eles dividem-sem em "Fachada" (pelo sufixo `Facade`) e Servicos (pelo sufixo `Service`). EJBs de fachada encapsulam todas as operacoes de um modulo do sistema. Em outras palavras, um modulo nao deve ser acessado de outra forma que nao pelo seu EJB de Fachada. EJBs de fachada nao contem regras de negocio, eles apenas delegam a execucao das opeeracoes para os EJBs de servico. EJBs de fachada sao sem estado por natureza (@Stateles), e elegiveis para oferecer servicos REST. Todos metodos nos EJBs de Fachada sao transacionais por natureza. Ja os EJBs de servico sao os responsaveis pela implementacao das regras de negocio do sistema. Eles somente podem ser acessados por EJBs de fachada (qualquer EJB de fachada) ou entao por outros EJBs de servico dentro do mesmo modulo. EJBs de servico podem acessar a camada de persistencia ou invocar operacoes em classes utilitarias ou sistemas externos. EJBs de servico nao devem realizar acesso direto ao banco de dados, exceto metodos marcados explicitamente pelos arquitetos da aplicacao. EJBs de servico sao por natureza sem estado (@Stateles). Todos os componentes Laranja executam dentro do servidor de aplicacao, e contam com recursos implicitos para log, persistencia, transacionamento, seguranca e escalabilidade.
  - **Amarelo**: Classes utilitarias da aplicacao, criadas para o proprio sistema ou utilizadas de frameworks e bibliotecas de terceiros. Deve-se evitar ao maximo a utilizacao de metodos estaticos em classes utiliarias. Quando isso for desejado, entao deve-ser utilizar o padrao Singleton.
  - **Lilas**: Sistema externo eh todo e qualquer componente, ferramenta ou aplicativo avesso ao Sistema EMapping que deve ser acessado para leitura ou escrita de informacoes. Somente EJBs de Servico ou entao Classes Utilitarias (Util) podem acessar um sistema externo, e nunca uma Action ou Facade.
  - **Marrom**: Persistencia, representada pelo JPA (JSR-317) pela implmentacao do Hibernate, oferece metodos para escrita e leitura de dados no banco de dados. Todas as operacoes realizadas sobre a camada de persistencia operam sobre o modelo de objetos da aplicacao, e nunca deve fazer acesso direto as tabelas do banco de dados. Em outras palavras, comandos SQL nao deve ser

executados sobre a camada de persistencia, mas sim comandos da EJB-QL. A camada de persistencia eh livre para realizar cache de objetos conforme indicacoes da equipe de arquitetura
    - **Vermelho**: Eh o banco de dados do sistema, representado por um SGBD relacional Oraclel ou Postres, e contem os objetos persistentes da aplicao, salvos em tabelas atraves do framework JPA da persistencia. Nenhum acesso direto da apliacao (codigo de programador) deve ocorrer nessa camada, exceto explicitamente demarcado pela equipe de arquitetura.

- Acoes:
  1. Toda iteracao do sistema inicia com uma solicitacao de usuario, que pode ser uma pessoa ou um sistema. A acao do usuario corre sempre sobre um navegador web. **Eh exatamente esta acao de usuario a chamada Feature do sistema, ou seja, uma funcao que o sistema oferece para o usuario**.
  2. A Action eh o ponto de entrada na aplicacao, ela representa a acao do usuario e pode executar operacoes simples como validacao de dados quanto ao tipo de dados, intervalo de valores e aplicacao de mascaras. Porem, fora isso, toda e qualquer regra de negocio do sistema eh despachada para o EJB de Fachada.
  3. Os EJBs de fachada sao objetos sem estado que simplesmente aglutinam os diversos servicos oferecidos por um modulo do sistema em uma inteface comum, que pode ser acessada pelo meio externo. Eles nao implementam nenhuma regra de negocio, apenas delegam as chamadas das Actions para os respectivos servicos de negocio. EJBs de fachada possuem caracteristicas de seguranca, transacao, log e auditoria implicitamente.
  4. Os EJBs de servico sao os principais elementos da aplicacao. Eles contem as regras de negocio do sistema, e sao acionados pelos EJBs de fachada.
  5. Quando necessario, um EJB de servico pode invocar operacoes em outros EJBs de servico dentro do mesmo modulo para complementar a regra de negocio necessaria.
  6. Quando necessario, um EJB de servico pode invocar operacoes em classes utilitarias da aplicacao, tanto criadas pelo programador (vide classe utilitaria DependencyBuilder@emapping) quanto de frameworks e bibliotecas de terceiros (vide classe utilitaria org.apache.commons.lang3.StringUtils@google).
  7. Nos casos onde um EJB de servico precisa invocar operacoes de regras de negocio de EJBs de servico que estao em outro modulo, ele precisa obrigatoriamente fazer isso atraves do EJB de fachada do respectivo modulo. Assim, nunca EJBs de servico de modulos distintos devem ter dependencias diretas entre si.
  8.
  9.
  10.
  11.
  12.

Implantacao