MAS - Modelo Arquitetural do Sistema

Nesta pagina

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

CONTEUDO

Tecnologias

Tecnologia	Versao	Usada para
Java	1.6	Linguagem utilizada no desenvolvimento de plugins Atlassian
Groovy	2.0.2	Utilizada para criar diferentes renders de conteúdo baseado nas features de builders oferecidas pela linguagem
Jersey	1.0.2	Implementação da JSR-311, criação de serviços REST
JAX-B	2.1	Marshalling e unmarshalling de JSON e XML
Atlassian SDK	4.1.4	SDK que expôe a API dos produtos Atlassian
Graphviz	2.30	Biblioteca para criar imagens de workflows
Wijmo	2.3.6	Biblioteca JS de widgets

Ferramentas

Ferramenta	Versao	Utilizadores	Usada para
Eclipse	Juno Release - Build id: 20120614-1722	Desenvolvedores	IDE de desenvolvimento
Balsamiq Mockups	Next.558 - 03/14/2011 12:18	Analistas, Projetistas, Arquitetos	Criação de mockups

Camadas da Aplicacao

? Anexo desconhecido Interface do Usuario

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

Camada de Servicos

Todos os serviços disponiveis sao implementados utilizando o padrao de REST da Atlassian. Esta camada oferece uma forma de comunicação entre as diferentes partes do plugin. Desta forma, a parte do Jira (o plugin que sera desenvolvido para o Jira) podera realizar operacoes para remover, atualizar, inserir e editar conteudos no Confluence.

Modelo de URI utilizado pela API REST da Atlassian e suas extensoes:

http://{host}:{port}/{context}/rest/autodoc/{version}/{operation}

- {host} e {port} definem onde a aplicacao esta rodando
- {context} é o Servlet Context da aplicacao. Por exemplo, no confluence normalmente é /confluence
- {operation} é a operação a ser executada
- {version} é a versao da API. Para utilizar a ultima versao da API eh possivel utilizar como valor /latest

Modelo de Dados

Atualmente a API disponibilizada pelo plugin do lado do Confluence aborda apenas a comunicacao utilizando JSON. Para ilustrar o modelo de dados, abaixo segue um exemplo de uma requisicao JSON.

```
{
   "workflow":{
      "name":"",
      "description":"",
      "steps":[
         {
            "name":"",
            "description":"",
            "transitions":[
               {
                  "name":"",
                  "description":"",
                  "outboundStep":"",
                  "conditions":[
                     {
                        "name":"",
                        "description":""
                     }
                  ],
                  "validators":[
                     {
                        "name":"",
                        "description":""
                     }
                  ],
                  "postfunctions":[
                     {
                        "name":"",
                        "description":""
                     }
                  ]
               }
            ]
         }
      ]
   }
}
```

Mudancas

Esta requisicao eh apenas um exemplo, este modelo pode mudar devido as diferentes operacoes disponiveis pela API REST.

Servicos Externos

Google Translate

A lingua padrao para descricoes de Validators, Conditions e Post Functions eh Ingles. Sabendo disso a API pode enviar requisicoes para servicos online de traducao (como Google Translate) para realizar as traducoes da forma que seja possivel disponibilizar a documentacao em diferentes linguas.

Se esta feature for implementada utilizando este tipo de abordagem o plugin desenvolvido necessariamente deve ter acesso a web.

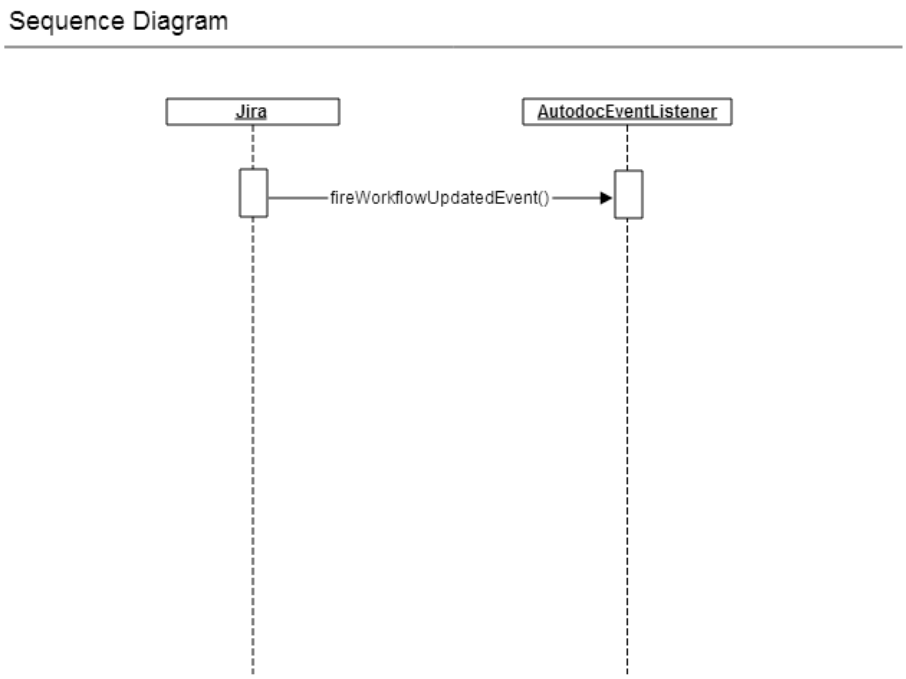
Funcionamento

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

Logico

Para o usuario final, o plugin realiza a tarefa de documentar as mudanças feitas nos workflows cadastrados no JIRA de forma automatica. No momento da mudança o plugin ira perceber esta mudança e alterar a documentacao disponivel no Confluence. Esta documentacao eh uma pagina que ate o presente momento (em proximas versoes poderemos ter novas features) tem um layout estatico por padrao. Classificamos por estatico porque o usuario nao ira alterar o layout do documento em si.

Segue abaixo um diagrama de sequencia para ilustrar esta funcionalidade.



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 -1) 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

Anexos

Especificacao de plugins REST da Atlassian: https://developer.atlassian.com/display/REST/REST+Plugin+Module#RESTPluginModule-PurposeoftheRESTPluginModule