Guia de desenvolvedor Keycloack

Prefácio

Em algumas das listas de exemplo, o que se pretende exibir em uma linha não se encaixa dentro da largura de página disponível. Estas linhas foram quebradas. Um '\' no final de uma linha significa que uma quebra foi introduzida para caber na página, com as seguintes linhas recuadas. Então:

```
Vamos fingir ter um extremamente \
longa linha que \
não se encaixa
Este é curto
```

É realmente:

```
Vamos fingir ter uma linha extremamente longa que não se encaixa
Este é curto
```

Admin REST API

Keycloak vem com uma API de administrador totalmente funcional com todos os recursos fornecidos pelo Console administrativo.

Para invocar a API, você precisa obter um token de acesso com as permissões apropriadas. As permissões necessárias estão descritas no Guia de Administração do Servidor.

Um token pode ser obtido permitindo autenticar seu aplicativo com Keycloak; consulte o Guia de Aplicações e Serviços de Garantia. Você também pode usar a concessão de acesso direto para obter um token de acesso.

Para obter documentação completa, consulte documentação da API.

Exemplos usando CURL

Autenticar com nome de usuário e senha

Obtenha token de acesso para usuário no mestre do reino com administrador de nome de usuário e senha de senha:

```
enrolar \
 -d "client_id=admin-cli" \
 -d "username=admin" \
 -d "password=password" \
 -d "grant_type=senha" \
 "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```

> Por padrão, este token expira em 1 minuto

O resultado será um documento JSON. Para invocar a API você precisa extrair o valor da propriedade access_token. Em seguida, você pode invocar a API, incluindo o valor no cabeçalho de autorização de solicitações à API.

O exemplo a seguir mostra como obter os detalhes do domínio mestre:

```
enrolar \
 -H "Autorização: portador eyJhbGciOiJSUz..." \
 "http://localhost:8080/auth/admin/realms/master"
```

Autenticar com uma conta de serviço

Antes de ser capaz de autenticar contra a API Admin REST usando um client_id e um client_secret você precisa ter certeza de que o cliente está configurado da seguinte forma:

- client_id é um cliente **confidencial** que pertence ao **mestre** do reino
- client_id tem opção habilitada para contas de serviço
- client_id tem um mapeador personalizado de "Audiência"
    - Público-cliente incluído: console de administração de segurança

Finalmente, verifique se client_id tem a função 'administrador' atribuída na guia "Funções de Conta de Serviço".

Depois disso, você poderá obter um token de acesso para a API Admin REST usando client_id e client_secret:

```
enrolar \
 -d "client_id=<YOUR_CLIENT_ID>" \
 -d "client_secret=<YOUR_CLIENT_SECRET>" \
 -d "grant_type=client_credentials" \
 "http://localhost:8080/auth/realms/master/protocol/openid-connect/token"
```

Exemplo usando Java

Há uma biblioteca de clientes Java para a API Admin REST que facilita o uso do Java. Para usá-lo a partir do seu aplicativo, adicione uma dependência da biblioteca `keycloak-admin-client`.

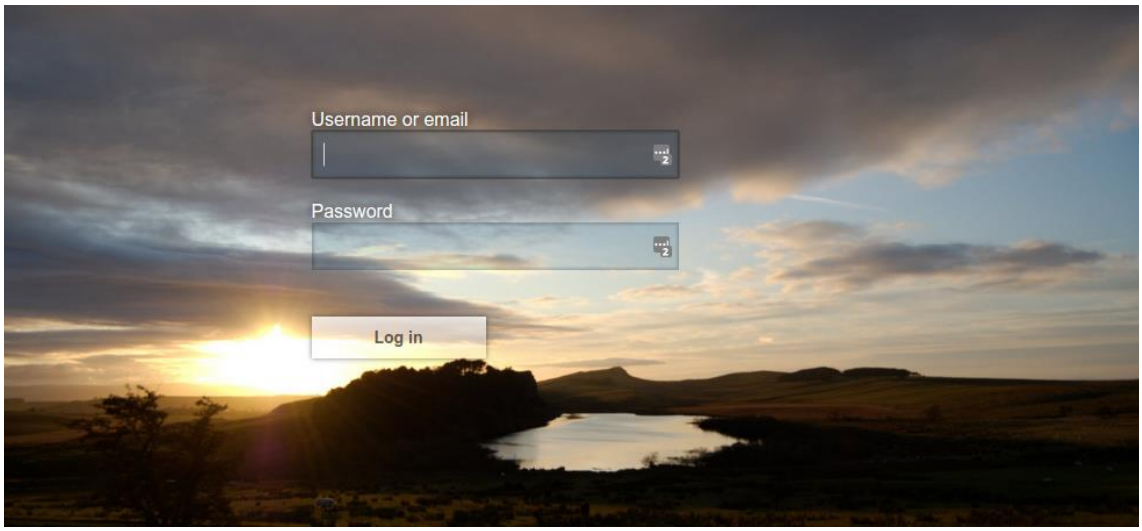
O exemplo a seguir mostra como usar a biblioteca de clientes Java para obter os detalhes do reino mestre:

```
import org.keycloak.admin.client.Keycloak;
import org.keycloak.representações.idm.RealmRepresentation;
...

Keycloak keycloak = Keycloak.getInstance(
"http://localhost:8080/auth",
"mestre",
"administrador",
"senha",
   "admin-cli");
ReinomRepresentação realm = keycloak.realm("mestre").toRepresentation();
```

Javadoc completo para o cliente administrador está disponível na Documentação API.

Temas

Keycloak fornece suporte temático para páginas da Web e e-mails. Isso permite personalizar a aparência e a sensação das páginas voltadas para o usuário final para que possam ser integradas com seus aplicativos.



Página de login com tema do exemplo do nascer do sol

Tipos temáticos

Um tema pode fornecer um ou mais tipos para personalizar diferentes aspectos do Keycloak. Os tipos disponíveis são:

- Conta - Gerenciamento de contas
- Administrador - Console administrativo
- E-mail - E-mails
- Login - Formulários de login
- Bem-vindo - Página de boas-vindas

Configure o tema

Todos os tipos de temas, exceto bem-vindos, são configurados através do Console administrativo. Para alterar o tema usado para um reino abra o Console de Administração,selecione seu reino na caixa de esquerda no canto superioresquerdo. Em Configurações de Reino clique em Temas.

> Para definir o tema para o console de administração mestre, você precisa definir o tema do console administrativo para o reino mestre. Para ver as alterações no console de administração, atualize a página.

Para alterar o tema de boas-vindas você precisa editar autônomo.xml, autônomo-ha.xmlou domínio.xml.

Adicione bem-vindoEle ao elemento tema, por exemplo:

```
<>
   ...
<começaMetema>custom</welcomeTheme>
   ...
</tema>
```

Se o servidor estiver em execução, você precisará reiniciar o servidor para que as alterações no tema de boas-vindas entrem em vigor.

Temas Padrão

O Keycloak vem empacotado com temas padrão no diretório de temas raiz do servidor. Para simplificar a atualização, você não deve editar os temas empacotados diretamente. Em vez disso, crie seu próprio tema que estende um dos temas empacotados.

Criando um tema

Um tema consiste em:

- Modelos HTML(Modelos freemarker)
- Imagens
- Pacotes de mensagens
- Folhas de estilo
- Scripts
- Propriedades temáticas

A menos que você planeje substituir cada página, você deve estender outro tema. Provavelmente você vai querer estender o tema Keycloak, mas você também pode considerar estender o tema base se você estiver mudando significativamente a aparência das páginas. O tema base consiste principalmente de modelos HTML e pacotes de mensagens, enquanto o tema Keycloak contém principalmente imagens e folhas de estilo.

Ao estender um tema, você pode substituir recursos individuais (modelos, folhas de estilo, etc.). Se você decidir substituir modelos HTML tenha em mente que você pode precisar atualizar seu modelo personalizado ao atualizar para uma nova versão.

Ao criar um tema, é uma boa ideia desativar o cache, pois isso torna possível editar recursos temáticos diretamente do diretório de temas sem reiniciar o Keycloak. Para fazer esta edição autônoma.xml. Para o tema definir estáticaMaxAge para -1 e ambos cacheTemplates e cacheThemes para falso:

```
<>
<stastáticaMaxAge>-1</estáticaMaxAge>
<cacheThemes>false</cacheThemes>
<cacheTemplates>false</cacheTemplates>
  ...
</tema>
```

Lembre-se de re-habilitar o cache na produção, pois isso afetará significativamente o desempenho.

Para criar um novo tema comece criando um novo diretório no diretório de temas. O nome do diretório se torna o nome do tema. Por exemplo, criar um tema chamado mytheme crie os temas do diretório/mito.

Dentro do diretório temático crie um diretório para cada um dos tipos que seu tema vai fornecer. Por exemplo, para adicionar o tipo de login ao tema mytheme crie os temas do diretório/mytheme/login.

Para cada tipo, crie um tema de arquivo.propriedades que permite definir alguma configuração para o tema. Por exemplo, para configurar os temas temáticos/mytheme/login que acabamos de criar para estender o tema base e importar alguns recursos comuns criar os temas de arquivo/mitoeme/login/theme.properties com seguintes conteúdos:

```
parent=base
importação=comum/keycloak
```

Agora você criou um tema com suporte para o tipo de login. Para verificar se ele funciona abra o console administrativo. Selecione seu reino e clique em `Temas`. Para o `Tema de Login`, selecione `mytheme` e clique em `Salvar`. Em seguida, abra a página de login para o reino.

Você pode fazer isso acessando através do seu aplicativo ou abrindo o console de Gerenciamento de Contas (`/realms/{realm name}/account`).

Para ver o efeito de alterar o tema pai, defina `parent=keycloak` em `theme.properties` e atualize a página de login.

Propriedades temáticas

As propriedades temáticas são definidas no arquivo `<THEME TYPE>/theme.properties` no diretório temático.

- pai - Tema dos pais para estender
- importação - Importação de recursos de outro tema
- estilos - Lista de estilos separados pelo espaço para incluir
- locais - Lista separada por comímas de locais suportados

Há uma lista de propriedades que podem ser usadas para alterar a classe css usada para determinados tipos de elementos. Para obter uma lista dessas propriedades, consulte o arquivo tema.properties no tipo correspondente do tema keycloak(`temas/keycloak/<THEME TYPE>/theme.properties`).

Você também pode adicionar suas próprias propriedades personalizadas e usá-las a partir de modelos personalizados.

Ao fazer isso, você pode substituir propriedades do sistema ou variáveis do ambiente usando esses formatos:

- `${some.system.property}` - para propriedades do sistema
- `${env. ENV_VAR}` - para variáveis ambientais.

Um valor padrão também pode ser fornecido no caso de a propriedade do sistema ou a variável ambiente não ser encontrada com `${foo:defaultValue}`.

> Se nenhum valor padrão for fornecido e não houver uma variável de propriedade ou ambiente correspondente do sistema, então nada será substituído e você acaba com o formato em seu modelo.

Aqui está um exemplo do que é possível:

```
javaVersion=${java.version}

unixHome=${env. HOME:Casa unix não encontrada}
windowsHome=${env. HOMEPATH:Casa do Windows não encontrada}
```

Folhas de estilo

Um tema pode ter uma ou mais folhas de estilo. Para adicionar uma folha de estilo, crie um arquivo no diretório <THEME TYPE>/resources/css do seu tema. Em seguida, adicione-o à propriedade estilos em tema.properties.

Por exemplo, para adicionar estilos.css ao mito criar temas/mytheme/login/resources/css/styles.css com o seguinte conteúdo:

```
.login-pf corpo {
fundo: DimGreynenhum;
}
```

Em seguida, edite temas/mytheme/login/theme.properties e adicione:

```
estilos=css/estilos.css
```

Para ver as alterações abra a página de login para o seu reino. Você notará que os únicos estilos que estão sendo aplicados são os da sua folha de estilo personalizada. Para incluir os estilos do tema pai, você precisa carregar os estilos desse tema também. Faça isso editando temas/mytheme/login/theme.properties e alterando estilos para:

```
styles=web_modules/@fortawesome/fontawesome-free/css/icons/all.css
web_modules/@patternfly/react-core/dist/styles/base.css
web_modules/@patternfly/react-core/dist/styles/app.css
node_modules/patternfly/dist/css/patternfly.min.css
node_modules/patternfly/dist/css/patternfly-additions.min.css css/login.css css/styles.css
```

> Para substituir estilos das folhas de estilo dos pais é importante que sua folha de estilos seja listada por último.

Scripts

Um tema pode ter um ou mais scripts, para adicionar um script criar um arquivo no diretório <THEME TYPE>/resources/js do seu tema. Em seguida, adicione-o à propriedade scripts em theme.properties.

Por exemplo, para adicionar script.js ao mito criar temas/mytheme/login/resources/js/script.js com o seguinte conteúdo:

```
alerta('Olá');
```

Em seguida, edite temas/mytheme/login/theme.properties e adicione:

```
scripts=js/script.js
```

Imagens

Para disponibilizar imagens ao tema, adicione-as ao diretório <THEME TYPE>/resources/img do seu tema. Estes podem ser usados de dentro de folhas de estilo ou diretamente em modelos HTML.

Por exemplo, para adicionar uma imagem ao mitoeme copiar uma imagem para temas/mitos/login/recursos/img/imagem.jpg.

Em seguida, você pode usar esta imagem de dentro de uma folha de estilo personalizada com:

```
corpo {
imagem de fundo: url('.. /img/imagem.jpg');
tamanho de fundo: cobertura;
}
```

Ou usar diretamente em modelos HTML adicione o seguinte a um modelo HTML personalizado:

```
<img src="${url.resourcesPath}/img/image.jpg">
```

Mensagens

O texto nos modelos é carregado a partir de pacotes de mensagens. Um tema que estende outro tema herdará todas as mensagens do pacote de mensagens dos pais e você pode substituir mensagens individuais adicionando <TEME TYPE>/messages/messages_en.properties ao seu tema.

Por exemplo, substituir o nome de usuário no formulário de login com o seu nome de usuário para o mito criar os temas de arquivo/mytheme/login/messages/messages_en.properties com o seguinte conteúdo:

```
nome de usuárioOrEmail=Seu nome de usuário
```

Dentro de uma mensagem, valores como {0} e {1} são substituídos por argumentos quando a mensagem é usada. Por exemplo, {0} login para {0} é substituído pelo nome do reino.

Os textos desses pacotes de mensagens podem ser substituídos por valores específicos do reino. Os valores específicos do reino são gerenciáveis via Interface do Usuário e API.

internacionalização

Keycloak apoia a internacionalização. Para permitir a internacionalização para um reino, consulte o Guia de Administração do Servidor. Esta seção descreve como você pode adicionar seu próprio idioma.

Para adicionar um novo idioma, crie o arquivo <THEME TYPE>/messages/messages_<LOCALE>.properties no diretório do seu tema. Em seguida, adicione-o à propriedade local em <THEME TYPE>/theme.properties. Para que um idioma esteja disponível para os usuários, o tema de login, conta e e-mail tem que suportar o idioma, então você precisa adicionar seu idioma para esses tipos de tema.

Por exemplo, para adicionar traduções norueguesas ao tema mytheme crie os temas do arquivo/mytheme/login/messages/messages_no.properties com o seguinte conteúdo:

```
usernameOrEmail=Brukernavn
senha=Passord
```

Todas as mensagens para as que você não fornece tradução usarão a tradução padrão em inglês.

Em seguida, edite temas/mytheme/login/theme.properties e adicione:

```
locales=en,não
```

Você também precisa fazer o mesmo para os tipos de conta e tema de e-mail. Para fazer isso, crie temas/mytheme/account/messages/messages_no.properties e themes/mytheme/email/messages/messages_no.properties. Deixar esses arquivos vazios resultará no uso das mensagens em inglês. Em seguida, copie temas/mytheme/login/theme.properties para temas/mytheme/account/theme.properties e themes/mytheme/email/theme.properties.

Finalmente, você precisa adicionar uma tradução para o seletor de idiomas. Isso é feito adicionando uma mensagem à tradução em inglês. Para isso, adicione os seguintes temas/mytheme/account/messages/messages_en.properties e themes/mytheme/login/messages/messages_en.properties:

```
locale_no=Norsk
```

Por propriedades de mensagem padrão, os arquivos devem ser codificados usando ISO-8859-1. Também é possível especificar a codificação usando um cabeçalho especial. Por exemplo, para usar a codificação UTF-8:

```
# codificação: UTF-8
usernameOrEmail=....
```

Consulte o Seletor de Locale sobre detalhes sobre como o local atual é selecionado.

Ícones de provedores de identidade personalizados

O Keycloak suporta a adição de ícones para provedores de identidade personalizados, que são exibidos na tela de login. Basta definir classes de ícones no seu arquivo tema de login.propriedades (ou seja, temas/mytheme/login/theme.properties) compadrão-chave kcLogoIdP-<alias>. Para o provedor de identidade com um pseudônimo myProvider, você pode adicionar uma linha, como abaixo, ao arquivo tema.properties do seu tema personalizado.

```
kcLogoIdP-myProvider = fa fa-lock
```

Todos os ícones estão disponíveis no site oficial do PatternFly4. Os ícones para provedores sociais já estão definidos em propriedades temáticas de login base(temas/keycloak/login/theme.properties), onde você pode se inspirar.

Modelos HTML

O Keycloak usa modelos freemarker para gerar HTML. Você pode substituir modelos individuais em seu próprio tema, criando <THEME TYPE>/<TEMPLATE>.ftl. Para obter uma lista de modelos utilizados, consulte temas/base/<THEME TYPE>.

Ao criar um modelo personalizado, é uma boa ideia copiar o modelo do tema base para o seu próprio tema e, em seguida, aplicar as modificações que você precisa. Tenha em mente ao atualizar para uma nova versão do Keycloak, você pode precisar atualizar seus modelos personalizados para aplicar alterações no modelo original, se aplicável.

Por exemplo, criar um formulário de login personalizado para os temas mitos copiar temas/base/login/login.ftl para temas/mitos/login e abri-lo em um editor. Depois da primeira linha (<#import ... >) adicione <h1>HELLO WORLD!</h1> assim:

```
< #import"template.ftl" como > de layout
<h1>HELLOWORLD! </h1>
...
```

Confira o Manual do FreeMarker para obter mais detalhes sobre como editar modelos.

E-mails

Para editar o assunto e o conteúdo para e-mails, por exemplo, e-mail de recuperação de senha, adicione um pacote de mensagens ao tipo de e-mail do seu tema. Há três mensagens para cada e-mail. Um para o sujeito, um para o corpo de texto simples e outro para o corpo html.

Para ver todos os e-mails disponíveis, dê uma olhada nos temas/base/e-mail/mensagens/messages_en.properties.

Por exemplo, alterar o e-mail de recuperação de senhas para o tema mytheme criar temas/mytheme/email/messages/messages_en.properties com o seguinte conteúdo:

```
passwordResetSubject=Minha recuperação de senha
passwordResetBody=Reset password link: {0}
senhaResetBodyHtml=<a href="{0}">Reset senha</a>
```

Implantação de Temas

Os temas podem ser implantados no Keycloak copiando o diretório temático para `temas` ou podem ser implantados como um arquivo. Durante o desenvolvimento você pode copiar o tema para o diretório de `temas`, mas na produção você pode querer considerar o uso de um `arquivo`. Um `arquivo` torna mais simples ter uma cópia versionada do tema, especialmente quando você tem várias instâncias de Keycloak, por exemplo, com clustering.

Para implantar um tema como um arquivo, você precisa criar um arquivo JAR com os recursos temáticos. Você também precisa adicionar um arquivo `META-INF/keycloak-themes.json` ao arquivo que lista os temas disponíveis no arquivo, bem como quais tipos cada tema fornece.

Por exemplo, para o tema `mitos` criar `mitos.jar` com os conteúdos:

- META-INF/keycloak-themes.json
- tema/mitoseme/login/theme.properties
- tema/mytheme/login/login.ftl
- tema/mytheme/login/resources/css/styles.css
- tema/mytheme/login/resources/img/image.png
- tema/mitoseme/login/mensagens/messages_en.propriedades
- tema/mitoseme/e-mail/mensagens/messages_en.propriedades

O conteúdo de `META-INF/keycloak-themes.json` neste caso seria:

```
{
   "temas": [{
"nome"  :  "mytheme",
"tipos":[  "login",  "e-mail"  ]
   }]
}
```

Um único arquivo pode conter vários temas e cada tema pode suportar um ou mais tipos.

Para implantar o arquivo no Keycloak basta colocá-lo no `diretório autônomo/implantações/diretório` do Keycloak e ele será carregado automaticamente.

Seletor de temas

Por padrão, o tema configurado para o reino é usado, com exceção de que os clientes podem substituir o tema de login. Esse comportamento pode ser alterado através do Seletor de Temas SPI.

Isso pode ser usado para selecionar diferentes temas para desktop e dispositivos móveis olhando para o cabeçalho do agente do usuário, por exemplo.

Para criar um seletor de temas personalizado, você precisa implementar o ThemeSelectorProviderFactory e ThemeSelectorProvider.

Siga os passos nas Interfaces do Provedor de Serviços para obter mais detalhes sobre como criar e implantar um provedor personalizado.

Recursos temáticos

Ao implementar provedores personalizados no Keycloak, muitas vezes pode haver a necessidade de adicionar modelos adicionais, recursos e pacotes de mensagens.

Um exemplo de exemplo de uso seria um autenticador personalizado que requer modelos e recursos adicionais.

A maneira mais fácil de carregar recursos temáticos adicionais é criar um JAR com modelos em recursos temáticos/modelos em pacotes de recursos/recursos temáticos e mensagens em recursos temáticos/mensagens e soltá-lo no diretório autônomo/implantações/diretório do Keycloak.

Se você quiser uma maneira mais flexível de carregar modelos e recursos que podem ser alcançados através do ThemeResourceSPI. Ao implementar o ThemeResourceProviderFactory e o ThemeResourceProvider, você pode decidir exatamente como carregar modelos e recursos.

Siga os passos nas Interfaces do Provedor de Serviços para obter mais detalhes sobre como criar e implantar um provedor personalizado.

Seletor de localidades

Por padrão, o local é selecionado usando o DefaultLocaleSelectorProvider que implementa a interface LocaleSelectorProvider. O inglês é a língua padrão quando a internacionalização é desativada. Com a internacionalização habilitada, o local é resolvido de acordo com a lógica descrita no Guia de Administração do Servidor.

Esse comportamento pode ser alterado através do LocaleSelectorSPI, implementando o LocaleSelectorProvider e o LocaleSelectorProviderFactory.

A interface LocaleSelectorProvider tem um único método, resolverLocale, que deve retornar um local dado um Modelo de Usuário RealmModel e um Modelo de Usuário nulo . A solicitação real está disponível no método KeycloakSession#getContext.

As implementações personalizadas podem estender o DefaultLocaleSelectorProvider para reutilizar partes do comportamento padrão. Por exemplo, para ignorar o cabeçalho de solicitação Aceitar-Idioma, uma implementação personalizada poderia estender o provedor padrão, substituir ele obterAceceptLanguageHeaderLocalee devolver um valor nulo. Como resultado, a seleção local cairá no idioma padrão dos reinos.

Siga os passos nas Interfaces do Provedor de Serviços para obter mais detalhes sobre como criar e implantar um provedor personalizado.

Atributos personalizados do usuário

Você pode adicionar atributos personalizados do usuário à página de registro e ao console de gerenciamento de contas com um tema personalizado. Este capítulo descreve como adicionar atributos a um tema personalizado, mas você deve se referir ao capítulo Temas sobre como criar um tema personalizado.

Página de Inscrição

Para poder inserir atributos personalizados na página de registro, copie os temas do modelo/base/login/register.ftl para o tipo de login do seu tema personalizado. Em seguida, abra a cópia em um editor.

Como exemplo para adicionar um número de celular à página de registro, adicione o seguinte trecho ao formulário:

```
<div class="form-group">
<div class ="${properties.kcLabelWrapperClass!}" >
<label para = classe"user.attributes.mobile" ="${properties.kcLabelClass!}" >número de
celular</etiqueta>
  </div>

<div class ="${properties.kcInputWrapperClass!}" >
<se tipode entrada= classe"texto" ="${propriedades.kcInputClass!}" id=
nome"user.attributes.mobile" = valor"user.attributes.mobile" =valor
"${(register.formData['user.attributes.mobile'!'!')}" />
  </div>
</div>
```

Certifique-se de que o nome do elemento html de entrada começa com os atributos do usuário. . No exemplo acima, o atributo será armazenado pelo Keycloak com o nome mobile.

Para ver as alterações, certifique-se de que seu reino esteja usando seu tema personalizado para o tema de login e abra a página de inscrição.

Console de gerenciamento de contas

Para poder gerenciar atributos personalizados na página do perfil do usuário no console de gerenciamento de conta, copie os temas do modelo/base/conta/conta/ftl para o tipo de conta do seu tema personalizado. Em seguida, abra a cópia em um editor.

Como exemplo de adicionar um número de celular à página da conta, adicione o seguinte trecho ao formulário:

```
<div class="form-group">
  <div class ="col-sm-2 col-md-2">
<label para =classe "user.attributes.mobile"  ="control-label">Número decelular</label>
  </div>

  <div class ="col-sm-10 col-md-10">
<se tipode entrada= classe"text" =id "form-control"  =nome "user.attributes.mobile"
=valor"user.attributes.mobile" = valor"${(account.attributes.mobile!''}" />
  </div>
</div>
```

Certifique-se de que o nome do elemento html de entrada começa com os atributos do usuário. .

Para ver as alterações, certifique-se de que seu reino esteja usando seu tema personalizado para o tema da conta e abra a página do perfil do usuário no console de gerenciamento da conta.

APIs de intermediação de identidade

Keycloak pode delegar autenticação a um IDP pai para login. Um exemplo típico disso é o caso em que você deseja que os usuários possam fazer login através de um provedor social como Facebook ou Google. O Keycloak também permite vincular contas existentes a um IDP intermediado. Esta seção fala sobre algumas APIs que seus aplicativos podem usar no que diz respeito à intermediação de identidade.

Recuperando tokens IDP externos

O Keycloak permite armazenar tokens e respostas do processo de autenticação com o IDP externo. Para isso, você pode usar a opção de configuração Store Token na página de configurações do IDP.

O código do aplicativo pode recuperar esses tokens e respostas para obter informações extras do usuário ou para invocar com segurança solicitações no IDP externo. Por exemplo, um aplicativo pode querer usar o token do Google para invocar outros serviços do Google e APIs REST. Para recuperar um token para um determinado provedor de identidade, você precisa enviar uma solicitação da seguinte forma:

```
GET /auth/realms/{realm}/broker/{provider_alias}/token HTTP/1.1
Anfitrião: localhost:8080
Autorização: Bearer <KEYCLOAK ACCESS TOKEN>
```

Um aplicativo deve ter autenticado com o Keycloak e ter recebido um token de acesso. Este token de acesso precisará ter o conjunto de tokens de leitura de função do cliente do corretor. Isso significa que o usuário deve ter um mapeamento de função para essa função e o aplicativo do cliente deve ter essa função dentro de seu escopo. Neste caso, dado que você está acessando um serviço protegido no Keycloak, você precisa enviar o token de acesso emitido pelo Keycloak durante a autenticação do usuário. Na página de configuração do corretor, você pode atribuir automaticamente essa função a usuários recém-importados ativando o switch Readable de Tokens Armazenados.

Esses tokens externos podem ser restabelecidos fazendo login novamente através do provedor ou usando a API de vinculação de conta iniciada pelo cliente.

Vinculação de conta iniciada pelo cliente

Alguns aplicativos querem se integrar com provedores sociais como o Facebook, mas não querem oferecer uma opção de login através desses provedores sociais. O Keycloak oferece uma API baseada em navegador que os aplicativos podem usar para vincular uma conta de usuário existente a um IDP externo específico. Isso é chamado de vinculação de conta iniciada pelo cliente. A vinculação da conta só pode ser iniciada por aplicativos OIDC.

A maneira como funciona é que o aplicativo encaminha o navegador do usuário para uma URL no servidor Keycloak solicitando que ele queira vincular a conta do usuário a um provedor externo específico (ou seja, facebook). O servidor inicia um login com o provedor externo. O navegador faz login no provedor externo e é redirecionado de volta para o servidor. O servidor estabelece o link e redireciona de volta para o aplicativo com uma confirmação.

Existem algumas pré-condições que devem ser atendidas pelo aplicativo do cliente antes que ele possa iniciar este protocolo:

- O provedor de identidade desejado deve ser configurado e habilitado para o reino do usuário no console administrativo.
- A conta de usuário já deve estar logada como um usuário existente através do protocolo OIDC
- O usuário deve ter uma `conta.manage-account` ou `account.manage-account-links de` mapeamento de papéis.
- O aplicativo deve ser concedido o escopo para essas funções dentro de seu token de acesso
- O aplicativo deve ter acesso ao seu token de acesso, pois precisa de informações dentro dele para gerar a URL redirecionar.

Para iniciar o login, o aplicativo deve fabricar uma URL e redirecionar o navegador do usuário para esta URL. A URL se parece com isso:

```
/{auth-server-root}/auth/realms/{realm}/broker/{provider}/link?client_id={id}&redirect_uri={uri}&nonce={nonce}&hash={hash}
```

Aqui está uma descrição de cada caminho e consulta param:

provedor

Este é o pseudônimo do provedor do IDP externo que você definiu na seção Provedor de `Identidade` do console administrativo.

client_id

Esta é a id cliente OIDC da sua aplicação. Quando você registrou o aplicativo como cliente no console administrativo, você teve que especificar esse id do cliente.

redirect_uri

Esta é a URL de retorno de chamada do aplicativo para a sua instituída após a criação do link da conta. Deve ser um padrão URI de redirecionamento de cliente válido. Em outras palavras, ele deve corresponder a um dos padrões de URL válidos que você definiu quando registrou o cliente no console administrativo.

Nonce

Esta é uma sequência aleatória que seu aplicativo deve gerar

hash

Este é um hash codificado por URL Base64. Este hash é gerado pela URL base64 codificando um hash SHA_256 de `nonce + token.getSessionState() + token.getIssuedFor() + provedor`. A variável token é obtida a partir do token de

acesso OIDC. Basicamente, você está hashing a nonce aleatória, o id de sessão do usuário, o id do cliente e o alias provedor de identidade que você deseja acessar.

Aqui está um exemplo do código Java Servlet que gera a URL para estabelecer o link da conta.

```
  KeycloakSecurityContext session = (KeycloakSecurityContext)
httpServletRequest.getAttribute(KeycloakSecurityContext.class.getName());
  AccessToken token = session.getToken();
  String clientId = token.getIssuedFor();
  String nonce = UUID.randomUUID().toString();
MessageDigest md = nulo;
  tente {
    md = MessageDigest.getInstance("SHA-256");
} catch  (NoSuchAlgorithmException e) {
    lançar novo RuntimeException(e);
  }
  Entrada de string = nonce + token.getSessionState() + clienteId + provedor;
  byte[] verificação = md.digest (input.getBytes(StandardCharsets.UTF_8));
  String hash = Base64Url.encode(check);
  request.getSession().setAttribute("hash",hash);
  Redirecionamento de cordasSUú = ...;
  Conta de stringLinkUrl = KeycloakUriBuilder.fromUri(authServerRootUrl)
              .path("/auth/realms/{realm}/broker/{provider}/link")
              .queryParam("nonce", nonce)
              .queryParam("hash", hash)
              .queryParam("client_id", clientId)
              .queryParam("redirect_uri",redirecionarUri).build(reino, provedor).toString();
```

Por que esse haxixe está incluído? Fazemos isso para que o servidor auth seja garantido para saber que o aplicativo do cliente iniciou a solicitação e nenhum outro aplicativo desonesto apenas pediu aleatoriamente que uma conta de usuário fosse vinculada a um provedor específico. O servidor auth verificará primeiro se o usuário está logado verificando o cookie SSO definido no login. Em seguida, ele tentará regenerar o hash com base no login atual e combiná-lo com o hash enviado pelo aplicativo.

Depois que a conta estiver vinculada, o servidor auth voltará para o `redirect_uri`. Se houver um problema no atendimento à solicitação de link, o servidor auth pode ou não redirecionar de volta para o `redirect_uri`. O navegador pode acabar em uma página de erro em vez de ser redirecionado de volta para o aplicativo. Se houver uma condição de erro e o servidor auth considerar seguro o suficiente para redirecionar para o aplicativo do cliente, um parâmetro adicional de consulta de `erro` será anexado ao `redirect_uri`.

> Embora esta API garanta que o aplicativo tenha iniciado a solicitação, ela não impede completamente os ataques de CSRF para esta operação. O aplicativo ainda é responsável por proteger contra alvos de ataques da CSRF em si mesmo.

Reestrações tokens externos

Se você estiver usando o token externo gerado fazendo login no provedor (ou seja, um token do Facebook ou GitHub), você pode atualizar esse token reiniciando a API de vinculação da conta.

Interfaces de provedor de serviços (SPI)

O Keycloak foi projetado para cobrir a maioria dos casos de uso sem exigir código personalizado, mas também queremos que seja personalizável. Para alcançar este Keycloak tem uma série de SPI (Service Provider Interfaces, interfaces de provedores de serviços) para as quais você pode implementar seus próprios provedores.

Implementação de um SPI

Para implementar um SPI, você precisa implementar suas interfaces ProviderFactory e Provider. Você também precisa criar um arquivo de configuração de serviço.

Por exemplo, para implementar o Seletor temático, você precisa implementar o ThemeSelectorProviderFactory e ThemeSelectorProvider e também fornecer o arquivo `META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory`.

Exemplo TemaSeletrorProviderFactory:

```
pacote org.acme.provider;

importar...

classe pública MyThemeSelectorProviderFactory implementa ThemeSelectorProviderFactory {

    @Override
tema públicoProvider(sessão KeycloakSession) {
retornar novo MyThemeSelectorProvider(sessão);
    }

    @Override
    vazio público init (Config.Scope config) {
    }

    @Override
    postinit de vazio público (fábrica KeycloakSessionFactory) {
    }

    @Override
    vazio público perto() {
    }
```

```
   @Override
público String getId() {
      retornar "myThemeSelector";
   }
}
```

> A Keycloak cria uma única instância de fábricas de provedores que torna possível armazenar o estado para várias solicitações. As instâncias do provedor são criadas chamando a criação na fábrica para cada solicitação, de modo que estes devem ser objeto de peso leve.

Exemplo ThemeSelectorProvider:

```
pacote org.acme.provider;

importar...

classe pública MyThemeSelectorProvider implementa ThemeSelectorProvider {

   público MyThemeSelectorProvider(Sessão KeycloakSession) {
   }


   @Override
cadeia pública getThemeName(Tipotema){
devolver "meu tema";
   }

   @Override
      vazio público perto() {
   }
}
```

Arquivo de configuração de serviço de exemplo (`META-INF/services/org.keycloak.theme.ThemeSelectorProviderFactory`):

```
org.acme.provider.MyThemeSelectorProviderFactory
```

Você pode configurar seu provedor através `de .xml autônomo`, `autônomo-ha.xml`ou `domínio.xml`.

Por exemplo, adicionando o seguinte a `autônomo.xml`:

```
nome <spi ="themeSelector">
<provo nome ="myThemeSelector"  ativado="verdadeiro">
      <propriedades>
<o nomeda propriedade= valor"tema" ="meu tema"/>
      </propriedades>
   </provedor>
```

```
</spi>
```

Em seguida, você pode recuperar o config no método init `ProviderFactory`:

```
vazio público init (Config.Scope config) {
   Tema de stringNo = config.get("tema");
}
```

Seu provedor também pode procurar outros provedores, se necessário. por exemplo:

```
classe pública MyThemeSelectorProvider  implementa  ThemeSelectorProvider {

sessão de sessão de sessão de Keycloaksession privada;

   público MyThemeSelectorProvider (sessão KeycloakSession) {
      estasessão .session = sessão;
   }

   @Override
   cadeia pública getThemeName (Tipo tema.tipo) {
      sessão de retorno.getContext().getRealm().getLoginTheme();
   }
}
```

Mostre informações da sua implementação de SPI no console administrativo

Às vezes, é útil mostrar informações adicionais sobre seu Provedor a um administrador keycloak. Você pode mostrar informações de tempo de compilação do provedor (por exemplo, versão do provedor personalizado atualmente instalado), configuração atual do provedor (por exemplo, url do sistema remoto com o que seu provedor fala) ou alguma informação operacional (tempo médio de resposta do sistema remoto com o que seu provedor conversa). O console administrativo Keycloak fornece a página Informações do Servidor para mostrar esse tipo de informação.

Para mostrar informações do seu provedor, basta implementar `org.keycloak.provider.serverInfoAwareProviderFactory` interface em seu `ProviderFactory`.

Implementação de exemplo para `MyThemeSelectorProviderFactory` do exemplo anterior:

```
pacote org.acme.provider;

importar...

classe pública MyThemeSelectorProviderFactory implementa ThemeSelectorProviderFactory,
ServerInfoAwareProviderFactory  {
   ...
```

```
  @Override
Público Mapa<String,  String>  getOperationalInfo() {
Map<String,  String>  ret  = novo <> LinkedHashMap();
ret.put("nome do tema",  "meu tema");
retorno ret;
  }
}
```

Use provedores disponíveis

Na implementação do seu provedor, você pode usar outros provedores disponíveis no Keycloak. Os provedores existentes podem ser normalmente recuperados com o uso do KeycloakSession, que está disponível para o seu provedor, conforme descrito na seção Implementando um SPI.

Keycloak tem dois tipos de provedor:

- **Tipos de provedor de implementação única** - Pode haver apenas uma única implementação ativa do tipo de provedor específico no tempo de execução keycloak. Por exemplo, o HostnameProvider especifica o nome do host a ser usado pelo Keycloak e que é compartilhado para todo o servidor Keycloak. Portanto, pode haver apenas uma implementação única deste provedor ativo para o servidor Keycloak. Se houver várias implementações de provedores disponíveis para o tempo de execução do servidor, uma delas precisa ser especificada na configuração do subsistema keycloak no autônomo.xml como o padrão. Por exemplo, tais como:

```
<spi name="hostname">
  <se>efault</default-provider>
  ...
</spi>
```

O padrão de valor usado como o valor do provedor padrão deve corresponder ao ID devolvido pelo ProviderFactory.getId() da implementação da fábrica do provedor específico. No código, você pode obter o provedor como keycloakSession.getProvider(HostnameProvider.class)

- **Vários tipos de provedores de implementação** - Esses são tipos de provedores, que permitem várias implementações disponíveis e trabalhando juntos no tempo de execução do Keycloak. Por exemplo, o provedor EventListener permite ter várias implementações disponíveis e registradas, o que significa que um evento específico pode ser enviado a todos os ouvintes (jboss-logging, sysout etc). No código, você pode obter uma instância especificada do provedor, por exemplo, como session.getProvider (EventListener.class, "jboss-logging") . Você precisa especificar provider_id do provedor como o segundo argumento, pois pode haver várias instâncias desse tipo de provedor conforme descrito acima. O ID do provedor deve corresponder ao ID devolvido pelo ProviderFactory.getId() da

implementação da fábrica de provedores em particular. Alguns tipos de provedor podem ser recuperados com o uso do `ComponentModel` como segundo argumento e alguns (por exemplo `Autenticador`) ainda precisam ser recuperados com o uso do `KeycloakSessionFactory`. Não é recomendável implementar seus próprios provedores dessa forma, pois ele pode ser preterido no futuro.

Registrando implementações de provedores

Existem duas maneiras de registrar implementações de provedores. Na maioria dos casos, a maneira mais simples é usar a abordagem do implantador Keycloak, pois isso lida com uma série de dependências automaticamente para você. Ele também suporta implantação quente, bem como re-implantação.

A abordagem alternativa é implantar como um módulo.

Se você estiver criando um SPI personalizado, você precisará implantá-lo como um módulo, caso contrário, recomendamos usar a abordagem do implantador Keycloak.

Usando o Keycloak Deployer

Se você copiar seu frasco de provedor para o keycloak `autônomo/implantações/diretório`, seu provedor será automaticamente implantado. A implantação quente também funciona. Além disso, o frasco do seu provedor funciona de forma semelhante a outros componentes implantados em um ambiente WildFly, na qual eles podem usar instalações como a `estrutura de implantação jboss.xml` arquivo. Este arquivo permite configurar dependências em outros componentes e carregar frascos e módulos de terceiros.

Os frascos do provedor também podem ser contidos em outras unidades implantáveis, como EARs e WARs. Implantar com um EAR realmente torna muito fácil usar frascos de terceiros, pois você pode apenas colocar essas bibliotecas no `diretório lib/do` EAR.

Registre um provedor usando Módulos

Para registrar um provedor usando módulos, crie primeiro um módulo. Para fazer isso, você pode usar o script jboss-cli ou criar manualmente uma pasta dentro de `KEYCLOAK_HOME/módulos` e adicionar seu frasco e um `módulo.xml`. Por exemplo, para adicionar o provedor de exemplo sysout do ouvinte de eventos usando o script `jboss-cli` executar:

```
KEYCLOAK_HOME/bin/jboss-cli.sh --command="module add --name=org.acme.provider --resources=target/provider.jar --dependencies=org.keycloak.keycloak-core,org.keycloak.keycloak.keycloak-server-spi"
```

Ou para criá-lo manualmente, criar a pasta KEYCLOAK_HOME/módulos/org/acme/provider/main. Em seguida, copie o provedor.jar para esta pasta e crie o módulo.xml com o seguinte conteúdo:

```
<?xml versão="1.0" codificação="UTF-8"?>
<module xmlns="urn:jboss:module:1.3" nome="org.acme.provider">
  < recursos>
    <resource-root path ="provedor.jar"/>
  </recursos>
  <dependências>
    <module nome ="org.keycloak.keycloak-core"/>
    <module nome ="org.keycloak.keycloak-server-spi"/>
  </dependências>
</módulo>
```

Depois de criar o módulo, você precisa registrar este módulo com o Keycloak. Isso é feito editando a seção de subsistema de servidor keycloak de autônomo.xml, autônomo-ha.xmlou domínio.xml, eadicionando-o aos provedores:

```
<subssistema xmlns="urna:jboss:domínio:keycloak-server:1.1">
<se>auth</web-context>
  <providas>
<provider>module:org.keycloak.examples.event-sysout</provider>
  </provedores>
  ...
```

Desativando um provedor

Você pode desativar um provedor definindo o atributo habilitado para o provedor falso em autônomo.xml, autônomo-ha.xmlou domínio.xml. Por exemplo, desativar o provedor de cache do usuário Infinispan adicione:

```
<spi nome ="userCache">
<provo nome="infinispan" ativado="falso"/>
</spi>
```

Alavancando Java EE

Os provedores de serviços podem ser embalados em qualquer componente Java EE desde que você configure o arquivo META-INF/serviços corretamente para apontar para seus provedores. Por exemplo, se o seu provedor precisar usar bibliotecas de terceiros, você pode empacotar seu provedor dentro de um ouvido e armazenar essas bibliotecas de terceiros no diretório lib/do ouvido. Observe também que os frascos de provedor podem fazer uso da estrutura de implantação jboss.xml arquivo que EJBs, WARS e EARs podem usar em um ambiente WildFly. Consulte a documentação do WildFly para obter mais detalhes sobre este arquivo. Ele permite que você puxe em dependências externas entre outras ações de grãos finos.

As implementações do ProviderFactory são necessárias para serem objetos java simples. Mas, atualmente, também apoiamos a implementação de classes de provedores como EJBs stateful. É assim que você faria:

```
@Stateful
@Local(EjbExampleUserStorageProvider.class)
classe pública EjbExampleUserStorageProvider  implementa  UserStorageProvider,
      UserLookupProvider,
      UserRegistrationProvider,
      UserQueryProvider,
      CredencialInputUpdater,
      CredencialInputValidator,
      OnUserCache
{
   @PersistenceContext
Entidades protegidasManager em;

modelo de modelo de modelo de componente protegido;
sessão de sessão de sessão de keycloaksession protegida;

   conjunto de vazio públicoModel (modelo ComponentModel) {
      estemodelo .= modelo;
   }

   conjunto de vazio públicoSession (sessão KeycloakSession) {
      estasessão .session = sessão;
   }


   @Remove
   @Override
   vazio público perto() {
   }
...
}
```

Você tem que definir a @Local anotação e especificar sua classe de provedor lá. Se você não fizer isso, o EJB não irá proxy a instância do provedor corretamente e seu provedor não funcionará.

Você deve colocar a anotação @Remove no método próximo () do seu provedor. Se você não fizer isso, o feijão imponente nunca será limpo e você pode eventualmente ver mensagens de erro.

As implementações do ProviderFactory são necessárias para serem objetos java simples. Sua classe de fábrica realizaria uma busca JNDI do EJB Stateful em seu método de criação().

```
classe pública EjbExampleUserStorageProviderFactory
      implementa UserStorageProviderFactory<EjbExampleUserStorageProvider> {
```

```
   @Override
ejbexampleUserStorageProvider create (KeycloakSession session, ComponentModel model) {
      tente {
InicialContext ctx = novo  Texto Inicial();
         EjbExampleUserStorageProvider provider =
(EjbExampleUserStorageProvider)ctx.lookup(
                  "java:global/user-storage-jpa-example/" +
EjbExampleUserStorageProvider.class.getSimpleName());
         provider.setModel(modelo);
         provider.setSession(sessão);
provedor de retorno;
} captura  (Exceção e) {
         lançar novo RuntimeException(e);
      }
   }
```

Provedores JavaScript

O Keycloak tem a capacidade de executar scripts durante o tempo de execução, a fim de permitir que os administradores personalizem funcionalidades específicas:

- Autenticador
- Política JavaScript
- Mapeador de protocolo de conexão OpenID

Autenticador

Os scripts de autenticação devem fornecer pelo menos uma das seguintes funções: `autenticar(..)` , que é chamado de ação `Authenticator#authenticate(AuthenticationFlowContext)` `(..)` , que é chamado de `Authenticator#action(AuthenticationFlowContext)`

O Autenticador Personalizado deve pelo menos fornecer o `autenticado(..)` função. Você pode usar o script `javax.script.Vinculações` dentro do código.

`roteiro`

o `ScriptModel` para acessar metadados de script

`reino`

o `Modelo Real`

`utilizador`

o modelo de usuário atual

sessão

o KeycloakSession ativo

autenticaçãoSessão

o atual Modelo de AutenticaçãoSession

httpRequest

a atual org.jboss.resteasy.spi.HttpRequest

tora

a org.jboss.logging.Logger escopo para ScriptBasedAuthenticator

> Você pode extrair informações adicionais de contexto do argumento de contexto passado para a função de ação (contexto) autenticada(contexto).

```
AutenticaçãoFlowError = Java.type("org.keycloak.authentication.AuthenticationFlowError");

autenticação de função (contexto) {

LOG.info(script.name + " -> trace auth para: " + nome de usuário.user);

se ( user.username === "testador"
    && user.getAttribute("someAttribute")
&& user.getAttribute("someAttribute"("someValue")) {

    contexto.falha (AuthenticationFlowError.INVALID_USER);
retorno;
 }

 contexto.sucesso();
}
```

Crie um JAR com os scripts para implantar

> Os arquivos JAR são arquivos ZIP regulares com uma extensão .jar.

Para disponibilizar seus scripts ao Keycloak, você precisa implantá-los no servidor. Para isso, você deve criar um arquivo JAR com a seguinte estrutura:

```
META-INF/keycloak-scripts.json

meu autenticador de script.js
minha política de script.js
meu script-mapper.js
```

O `META-INF/keycloak-scripts.json` é um descritor de arquivos que fornece informações sobre metadados sobre os scripts que você deseja implantar. É um arquivo JSON com a seguinte estrutura:

```
{
"autenticadores":[
      {
"nome": "Meu Autenticador",
"fileName": "meu-script-autenticador.js",
"descrição": "Meu Autenticador de um arquivo JS"
      }
   ],
"políticas":[
      {
"nome": "Minha Política",
"fileName": "my-script-policy.js",
"descrição": "Minha política de um arquivo JS"
      }
   ],
"mappers"
      {
"nome": "Meu Mapper",
"fileName": "my-script-mapper.js",
"descrição": "Meu Mapper de um arquivo JS"
      }
   ]
}
```

Este arquivo deve fazer referência aos diferentes tipos de provedores de script que você deseja implantar:

- `Autenticadores`

  Para autenticadores de script openID Connect. Você pode ter um ou vários autenticadores no mesmo arquivo JAR

- `Políticas`

  Para Políticas JavaScript ao usar os Serviços de Autorização keycloak. Você pode ter uma ou várias políticas no mesmo arquivo JAR

- `Mappers`

  Para mapeador de protocolo de script openID connect. Você pode ter um ou vários mapeador no mesmo arquivo JAR

Para cada arquivo de script em seu arquivo `JAR`, você deve ter uma entrada correspondente em `META-INF/keycloak-scripts.json` que mapeia seus arquivos de scripts para um tipo específico de provedor. Para isso, você deve fornecer as seguintes propriedades para cada entrada:

- `nome`

Um nome amigável que será usado para mostrar os scripts através do Keycloak Administration Console. Se não for fornecido, o nome do arquivo de script será usado em vez disso

- `descrição`

  Um texto opcional que melhor descreve a intenção do arquivo de script

- `Filename`

  O nome do arquivo do script. Esta propriedade é **obrigatória** e deve mapear para um arquivo dentro do JAR.

Implantar o JAR de script

Uma vez que você tenha um arquivo JAR com um descritor e os scripts que deseja implantar, você só precisa copiar o JAR para o keycloak `autônomo/implantações/diretório`.

Usando o Keycloak Administration Console para carregar scripts

> A capacidade de carregar scripts através do console administrativo é preterida e será removida em uma versão futura do Keycloak

Os administradores não podem carregar scripts no servidor. Esse comportamento evita danos potenciais ao sistema no caso de scripts maliciosos serem executados acidentalmente. Os administradores devem sempre implantar scripts diretamente no servidor usando um arquivo JAR para evitar ataques quando você executa scripts no tempo de execução.

A capacidade de carregar scripts pode ser explicitamente habilitada. Isso deve ser usado com muito cuidado e planos devem ser criados para implantar todos os scripts diretamente no servidor o mais rápido possível.

Para obter mais detalhes sobre como ativar o recurso `upload_scripts`. Por favor, dê uma olhada nos perfis.

SPIs disponíveis

Se você quiser ver a lista de todos os SPIs disponíveis no tempo de execução, você pode verificar a página `Informações do Servidor` no console administrativo, conforme descrito na seção Admin Console.

Ampliação do Servidor

A estrutura keycloak SPI oferece a possibilidade de implementar ou substituir determinados provedores incorporados. No entanto, o Keycloak também fornece recursos para ampliar suas funcionalidades e domínios principais. Isso inclui possibilidades de:

- Adicione pontos finais de REST personalizados ao servidor Keycloak
- Adicione seu próprio SPI personalizado
- Adicione entidades JPA personalizadas ao modelo de dados Keycloak

Adicione pontos finais de REST personalizados

Esta é uma extensão muito poderosa, que permite que você implante seus próprios pontos finais REST para o servidor Keycloak. Ele permite todos os tipos de extensões, por exemplo, a possibilidade de acionar a funcionalidade no servidor Keycloak, que não está disponível através do conjunto padrão de pontos finais do Keycloak REST incorporados.

Para adicionar um ponto final DE REST personalizado, você precisa implementar as interfaces `RealmResourceProviderFactory` e `RealmResourceProvider`. O `RealmResourceProvider` tem um método importante:

```
Objeto getResource();
```

que permite que você devolva um objeto, que atua como um recurso JAX-RS. Para mais detalhes, consulte o Javadoc e nossos exemplos. Há um exemplo muito simples na distribuição de exemplos em `provedores/descanso` e há um exemplo mais avançado em `provedores/extensão dedomínio`, que mostra como adicionar um ponto final de REST autenticado e outras funcionalidades como adicionar seu próprio SPI ou estender o modelo de dados com suas próprias entidades JPA.

Para obter detalhes sobre como empacotar e implantar um provedor personalizado, consulte o capítulo Interfaces do Provedor de Serviços.

Adicione seu próprio SPI personalizado

Isso é útil especialmente com os pontos finais custom REST. Para adicionar seu próprio tipo de SPI, você precisa implementar a interface `org.keycloak.provider.Spi` e definir o ID do seu SPI e as classes `ProviderFactory` e `Provider`. Isso se parece com isso:

```
pacote org.keycloak.examples.domainextension.spi;

importar...

classe pública ExampleSpi implementa Spi {

  @Override
  público booleano éInternal() {
```

```
        retornofalso;
    }

    @Override
    cadeia pública getName() {
devolver "exemplo";
    }

    @Override
classe pública<? estende  Provedor> getProviderClass() {
        retorno ExampleService.class;
    }

    @Override
@SuppressWarnings ("rawtypes")
classe pública<? estende  ProviderFactory> getProviderFactoryClass() {
        retorno ExemploServiceProviderFactory.class;
    }

}
```

Em seguida, você precisa criar o arquivo `META-INF/services/org.keycloak.provider.Spi` e adicionar a classe do seu SPI a ele. Por exemplo:

```
org.keycloak.examples.domainextension.spi.ExampleSpi
```

O próximo passo é criar as interfaces `ExampleServiceProviderFactory`, que se estende a partir do `ProviderFactory` e `ExampleService`, que se estendedo `Provedor`. O `ExampleService` geralmente contém os métodos de negócios necessários para o seu caso de uso. Observe que a instância `ExampleServiceProviderFactory` é sempre escopo por aplicativo, no entanto, o `ExampleService` é escopo por solicitação (ou mais precisamente por ciclo de vida `keycloakSession`).

Finalmente, você precisa implementar seus provedores da mesma forma descrita no capítulo Interfaces do Provedor de Serviços.

Para obter mais detalhes, dê uma olhada na distribuição de exemplo em `provedores/extensão de domínio`,que mostra um SPI de exemplo semelhante ao acima.

Adicione entidades JPA personalizadas ao modelo de dados Keycloak

Se o modelo de dados Keycloak não corresponder exatamente à solução desejada ou se você quiser adicionar alguma funcionalidade principal ao Keycloak, ou quando você tiver seu próprio ponto final REST, você pode querer estender o modelo de dados Keycloak. Nós permitimos que você adicione suas próprias entidades JPA ao Keycloak JPA `EntityManager` .

Para adicionar suas próprias entidades JPA, você precisa implementar `jpaEntityProviderFactory` e `JpaEntityProvider`. O `JpaEntityProvider` permite que você devolva uma lista de suas entidades JPA personalizadas e forneça a localização e a id do changelog liquibase. Uma implementação de exemplo pode ser assim:

> Esta é uma API não suportada, o que significa que você pode usá-la, mas não há garantia de que ela não será removida ou alterada sem aviso prévio.

```
classe pública ExemploJpaEntityProvider implementa JpaEntityProvider {

  Lista de suas entidades JPA.
  @Override
Lista pública<Class<?>> getEntities() {
    retorno Coleções.<Class<?>>singletonList (Empresa.class);
  }

  Isso é usado para retornar a localização do arquivo de changelog liquibase.
  Você pode retornar nulo se não quiser que o Liquibase crie e atualize o esquema DB.
  @Override
  cadeia pública getChangelogLocation() {
      retorno "META-INF/exemplo-changelog.xml";
  }

  Método auxiliar, que será usado internamente pela Liquibase.
  @Override
  cadeia pública getFactoryId() {
    devolver "amostra";
  }

  ...
}
```

No exemplo acima, adicionamos uma única entidade JPA representada pela `empresa` declasse. No código do seu ponto final REST, você pode usar algo assim para recuperar o `EntityManager` e chamar as operações DB nele.

```
EntityManager em = session.getProvider(JpaConnectionProvider.class).getEntityManager();
Empresa myCompany = em.find (Empresa.class, "123");
```

Os métodos `getChangelogLocation` e `getFactoryId` são importantes para suportar a atualização automática de suas entidades pela Liquibase. Liquibase é uma estrutura para atualizar o esquema de banco de dados, que o Keycloak usa internamente para criar o esquema DB e atualizar o esquema DB entre as versões. Você pode precisar usá-lo também e criar um changelog para suas entidades. Observe que a versão do seu próprio changelog Liquibase é independente das versões do Keycloak. Em outras palavras, quando você atualiza para uma nova versão keycloak, você não é forçado a atualizar seu esquema ao mesmo tempo. E vice-versa, você pode atualizar seu esquema mesmo sem atualizar a versão Keycloak. A atualização do Liquibase é sempre feita na inicialização do servidor, então para acionar uma atualização db do seu esquema, basta adicionar o novo changeet ao seu arquivo de changelog Liquibase (no exemplo acima é o

arquivo `META-INF/exemplo-changelog.xml` que deve ser embalado no mesmo JAR que as entidades JPA e `ExampleJpaEntityProvider`) e, em seguida, reiniciar oservidor. O esquema DB será atualizado automaticamente na inicialização.

Para obter mais detalhes, dê uma olhada na distribuição de exemplo em `provedores de exemplo/extensão dedomínio`, que mostra o `ExemploJpaEntityProvider` e o `exemplo-changelog.xml` descrito acima.

> Não se esqueça de sempre fazer backup do seu banco de dados antes de fazer qualquer alteração no changelog liquibase e desencadear uma atualização DB.

SPI de autenticação

Keycloak inclui uma série de diferentes mecanismos de autenticação: kerberos, senha, otp e outros. Esses mecanismos podem não atender a todos os seus requisitos e você pode querer conectar seus próprios costumes. O Keycloak fornece um SPI de autenticação que você pode usar para escrever novos plugins. O console administrativo suporta aplicar, encomendar e configurar esses novos mecanismos.

Keycloak também suporta um formulário de registro simples. Diferentes aspectos deste formulário podem ser ativados e desativados, ou seja, o suporte à recaptcha pode ser desligado e ligado. O SPI de autenticação da mesma autenticação pode ser usado para adicionar outra página ao fluxo de registro ou reimplementá-lo inteiramente. Há também um SPI de grãos finos adicionais que você pode usar para adicionar validações específicas e extensões do usuário ao formulário de registro incorporado.

Uma ação necessária no Keycloak é uma ação que um usuário tem que realizar depois de autenticar. Depois que a ação é realizada com sucesso, o usuário não precisa realizar a ação novamente. Keycloak vem com algumas ações incorporadas, como "redefinir senha". Essa ação obriga o usuário a alterar sua senha depois de ter logado. Você pode escrever e conectar suas próprias ações necessárias.

> Se o autenticador ou a implementação de ação necessária estiver usando alguns atributos do usuário como atributos de metadados para vincular/estabelecer a identidade do usuário, certifique-se de que os usuários não são capazes de editar os atributos e os atributos correspondentes são somente leitura. Veja os detalhes no capítulo de mitigaçãodo modelo Ameaça.

termos

Para primeiro aprender sobre o SPI de Autenticação, vamos passar por cima de alguns dos termos usados para descrevê-lo.

Fluxo de autenticação

Um fluxo é um contêiner para todas as autenticações que devem acontecer durante o login ou registro. Se você for para a página de autenticação do console administrativo, você pode visualizar todos os fluxos definidos no sistema e de quais autenticadores eles são compostos. Os fluxos podem conter outros fluxos. Você também pode vincular um novo fluxo diferente para login do navegador, acesso direto de subvenção e registro.

Autenticador

Um autenticador é um componente plugável que mantém a lógica para realizar a autenticação ou ação dentro de um fluxo. Geralmente é um singleton.

execução

Uma execução é um objeto que liga o autenticador ao fluxo e ao autenticador à configuração do autenticador. Os fluxos contêm entradas de execução.

Requisito de execução

Cada execução define como um autenticador se comporta em um fluxo. O requisito define se o autenticador está habilitado, desativado, condicional, necessário ou uma alternativa. Um requisito alternativo significa que o autenticador é suficiente para validar o fluxo em que está, mas não é necessário. Por exemplo, no fluxo incorporado do navegador, a autenticação de cookies, o Redirecionador do Provedor de Identidade e o conjunto de todos os autenticadores no subfluxo de formulários são todas alternativas. À medida que são executados em uma ordem sequencial de cima para baixo, se um deles for bem sucedido, o fluxo é bem sucedido, e qualquer execução seguinte no fluxo (ou sub-fluxo) não é avaliada.

Autenticador Config

Este objeto define a configuração do Autenticador para uma execução específica dentro de um fluxo de autenticação. Cada execução pode ter um config diferente.

Ação Necessária

Após a conclusão da autenticação, o usuário pode ter uma ou mais ações únicas que ele deve concluir antes de poder fazer login. O usuário pode ser obrigado a configurar um gerador de token OTP ou redefinir uma senha expirada ou até mesmo aceitar um documento de Termos e Condições.

Visão geral do algoritmo

Vamos falar sobre como tudo isso funciona para o login do navegador. Vamos supor os seguintes fluxos, execuções e sub-fluxos.

```
Cookie - ALTERNATIVA
Kerberos - ALTERNATIVA
Subfluxo de formulários - ALTERNATIVA
        Formulário de nome de usuário/senha - OBRIGATÓRIO
        Subfluxo condicional OTP - CONDICIONAL
                Condição - Usuário Configurado - OBRIGATÓRIO
                Formulário OTP - OBRIGATÓRIO
```

No nível superior da forma temos 3 execuções das quais todas são alternativamente necessárias. Isso significa que se algum destes for bem sucedido, então os outros não têm que executar. O formulário Nome de Usuário/Senha não é executado se houver um conjunto de cookies SSO ou um login Kerberos bem-sucedido. Vamos percorrer os passos de quando um cliente redireciona pela primeira vez para keycloak para autenticar o usuário.

1. O provedor de protocolo OpenID Connect ou SAML desembala dados relevantes, verifica o cliente e quaisquer assinaturas. Ele cria um Modelo de Autenticação. Ele olha para o fluxo do navegador, então começa a executar o fluxo.
2. O fluxo olha para a execução do biscoito e vê que ele é uma alternativa. Ele carrega o provedor de biscoitos. Ele verifica se o provedor de cookies requer que um usuário já esteja associado à sessão de autenticação. O provedor de cookies não requer um usuário. Se isso acontecesse, o fluxo abortaria e o usuário veria uma tela de erro. O provedor de cookies então executa. Seu objetivo é ver se há um conjunto de cookies SSO. Se houver um conjunto, ele é validado e o UserSessionModel é verificado e associado ao Modelo de Autenticação. O provedor Cookie retorna um status de sucesso () se o cookie SSO existir e for validado. Uma vez que o provedor de cookies retornou o sucesso e cada execução neste nível do fluxo é ALTERNATIVA, nenhuma outra execução é executada e isso resulta em um login bem-sucedido. Se não houver nenhum cookie SSO, o provedor de cookies retorna com um status de tentativa(). Isso significa que não houve nenhuma condição de erro, mas nenhum sucesso também. O provedor tentou, mas o pedido não foi configurado para lidar com esse autenticador.
3. Em seguida, o fluxo olha para a execução Kerberos. Esta também é uma alternativa. O provedor kerberos também não exige que um usuário já esteja configurado e associado ao Modelo de Autenticação Para que este provedor seja executado. Kerberos usa o protocolo de navegador SPNEGO. Isso requer uma série de desafios/respostas entre o servidor e o cliente trocando cabeçalhos de negociação. O provedor kerberos não vê nenhum cabeçalho de negociação, por isso assume que esta é a primeira interação entre o servidor e o cliente. Portanto, cria uma resposta de desafio HTTP ao cliente e define um status forceChallenge(). Um forceChallenge() significa que essa resposta HTTP não pode ser ignorada pelo fluxo e deve ser devolvida ao

cliente. Se, em vez disso, o provedor retornasse um status de desafio(), o fluxo manteria a resposta do desafio até que todas as outras alternativas sejam tentadas. Então, nesta fase inicial, o fluxo pararia e a resposta ao desafio seria enviada de volta para o navegador. Se o navegador responder com um cabeçalho de negociação bem-sucedido, o provedor associa o usuário à AutenticaçãoSessão e o fluxo termina porque o resto das execuções neste nível do fluxo são todas alternativas. Caso contrário, novamente, o provedor kerberos define uma tentativa() status e o fluxo continua.

4. A próxima execução é um subfluxo chamado Forms. As execuções desse subfluxo são carregadas e ocorre a mesma lógica de processamento.

5. A primeira execução no subfluxo Formulários é o provedor UsernamePassword. Este provedor também não exige que um usuário já esteja associado ao fluxo. Este provedor cria uma resposta HTTP de desafio e define seu status para desafiar(). Esta execução é necessária, de modo que o fluxo honra este desafio e envia a resposta HTTP de volta para o navegador. Esta resposta é uma renderização da página HTML nome de usuário/senha. O usuário insere em seu nome de usuário e senha e clica em enviar. Esta solicitação HTTP é direcionada ao provedor UsernamePassword. Se o usuário digitar um nome de usuário ou senha inválido, uma nova resposta de desafio será criada e um status de failureChallenge() será definido para esta execução. Um failureChallenge() significa que há um desafio, mas que o fluxo deve registrar isso como um erro no registro de erro. Esse registro de erro pode ser usado para bloquear contas ou endereços IP que tiveram muitas falhas de login. Se o nome de usuário e senha forem válidos, o provedor associou o Modelo de Usuário ao Modelo de Autenticação e retorna um status de sucesso().

6. A próxima execução é um subfluxo chamado OTP Condicional. As execuções desse subfluxo são carregadas e ocorre a mesma lógica de processamento. Sua exigência é condicional. Isso significa que o fluxo avaliará primeiro todos os executores condicionales que ele contém. Executores condicionais são autenticadores que implementam CondicionalAuthenticator, e devem implementar o método boolean matchCondition (AuthenticationFlowContext context). Um subfluxo condicional chamará o método matchCondition de todas as execuções condicionais que contém, e se todas elas avaliarem como verdadeiras, agirá como se fosse um subfluxo necessário. Se não, agirá como se fosse um subfluxo desativado. Autenticadores condicionais são usados apenas para este fim, e não são usados como autenticadores. Isso significa que, mesmo que o autenticador condicional avalie como "verdadeiro", então isso não marcará um fluxo ou subfluxo como bem sucedido. Por exemplo, um fluxo contendo apenas um subfluxo condicional com apenas um autenticador condicional nunca permitirá que um usuário faça login.

7. A primeira execução do subfluxo OTP Condicional é a Condição - Configuração do Usuário. Este provedor exige que um usuário tenha sido associado ao fluxo. Esse requisito está satisfeito porque o provedor

UsernamePassword já associou o usuário ao fluxo. O método de correspondência deste provedor avaliará o método configuradoPara todos os outros Autenticadores em seu subfluxo atual. Se o subfluxo contiver executores com o requisito definido como necessário, o método matchCondition só avaliará se todos os autenticadores necessários configurarem o método para ser real. Caso contrário, o método matchCondition avaliará se algum autenticador alternativo avaliará a verdade.

8. A próxima execução é o Formulário OTP. Este provedor também exige que um usuário tenha sido associado ao fluxo. Esse requisito está satisfeito porque o provedor UsernamePassword já associou o usuário ao fluxo. Uma vez que um usuário é necessário para este provedor, o provedor também é perguntado se o usuário está configurado para usar este provedor. Se o usuário não estiver configurado, então o fluxo configurará uma ação necessária que o usuário deve executar após a autenticação ser concluída. Para OTP, isso significa a página de configuração OTP. Se o usuário estiver configurado, ele será solicitado a inserir seu código otp. Em nosso cenário, devido ao subfluxo condicional, o usuário nunca verá a página de login OTP, a menos que o subfluxo Condicional OTP seja definido como Necessário.
9. Depois que o fluxo é concluído, o processador de autenticação cria um UserSessionModel e o associa ao Modelo de Autenticação. Em seguida, verifica se o usuário é obrigado a concluir quaisquer ações necessárias antes de fazer login.
10. Primeiro, o método de avaliação de cada ação necessária É chamado. Isso permite que o provedor de ação necessário descubra se há algum estado que possa desencadear a ação a ser disparada. Por exemplo, se o seu reino tiver uma política de expiração de senha, ela pode ser acionada por esse método.
11. Cada ação necessária associada ao usuário que tem seu método requeridoActionChallenge() chamado. Aqui, o provedor configura uma resposta HTTP que torna a página para a ação necessária. Isso é feito definindo um status de desafio.
12. Se a ação necessária for finalmente bem sucedida, a ação necessária será removida da lista de ações exigidas pelo usuário.
13. Depois que todas as ações necessárias foram resolvidas, o usuário é finalmente logado.

Autenticador SPI Walk Through

Nesta seção, vamos dar uma olhada na interface Autenticador. Para isso, vamos implementar um autenticador que exige que um usuário entre na resposta a uma pergunta secreta como "Qual é o nome de solteira de sua mãe?". Este exemplo é totalmente implementado e contido no diretório de exemplos/provedores/autenticadores da distribuição de demonstração do Keycloak.

Para criar um autenticador, você deve, no mínimo, implementar as interfaces org.keycloak.authentication.AuthenticatorFactory e Authenticator. A interface Autenticador define a lógica. O AuthenticatorFactory é responsável pela criação de instâncias de um Autenticador. Ambos estendem um conjunto de interfaces mais genéricas do Provedor e do ProviderFactory que outros componentes keycloak como a User Federation fazem.

Alguns autenticadores, como o CookieAuthenticator, não contam com uma credencial que o usuário tem ou sabe autenticar o usuário. No entanto, alguns autenticadores, como o autenticador PasswordForm ou o OTPFormAuthenticator, dependem do usuário inserindo algumas informações e verificando essas informações contra algumas informações no banco de dados. Para o PasswordForm, por exemplo, o autenticador verificará o hash da senha em um hash armazenado no banco de dados, enquanto o OTPFormAuthenticator verificará o OTP recebido contra o gerado a partir do segredo compartilhado armazenado no banco de dados.

Esses tipos de autenticadores são chamados de CredenciaisValidadores, e exigirão que você implemente mais algumas classes:

- Uma classe que estende org.keycloak.credencial.CredentialModel, e que pode gerar o formato correto da credencial no banco de dados
- Uma classe implementando o org.keycloak.credencia.credentialProvider e interface, e uma classe implementando sua interface de fábrica CredentialProviderFactory.

O SecretQuestionAuthenticator que veremos nesta caminhada é um CredenciamentoValidador, então vamos ver como implementar todas essas classes.

Aulas e implantação de embalagens

Você empacotará suas aulas dentro de um único frasco. Este frasco deve conter um arquivo chamado org.keycloak.authentication.AuthenticatorFactory e deve estar contido no META-INF/serviços/ diretório do seu frasco. Este arquivo deve listar o nome de classe totalmente qualificado de cada implementação AutenticadorFactory que você tem no frasco. Por exemplo:

```
org.keycloak.examples.authenticator.SecretQuestionAuthenticatorFactory
org.keycloak.examples.authenticator.AnotherProviderFactory
```

Este serviço/arquivo é usado pelo Keycloak para digitalizar os provedores que ele tem que carregar no sistema.

Para implantar este frasco, basta copiá-lo para o diretório de provedores.

Ampliando a classe CredentialModel

Em Keycloak, as credenciais são armazenadas no banco de dados na tabela Credenciais. Tem a seguinte estrutura:

```
----------------------------
| |
----------------------------
| user_ID |
----------------------------
| credential_type |
----------------------------
| created_date |
----------------------------
| user_label |
----------------------------
| secret_data |
----------------------------
| credential_data |
----------------------------
| | prioridade
----------------------------
```

onde:

- `ID` é a chave principal da credencial.
- `user_ID` é a chave estrangeira que liga a credencial a um usuário.
- `credential_type` é um conjunto de strings durante a criação que deve referenciar um tipo de credencial existente.
- `created_date` é o data-hora de criação (em formato longo) da credencial.
- `user_label` é o nome editável da credencial pelo usuário
- `secret_data` contém um json estático com as informações que não podem ser transmitidas fora de Keycloak
- `credential_data` contém um json com as informações estáticas da credencial que podem ser compartilhadas no console administrativo ou através da API REST.
- `prioridade` define como "preferido" uma credencial é para um usuário, para determinar qual credencial apresentar quando um usuário tem múltiplas escolhas.

Como os campos secret_data e credential_data são projetados para conter json, cabe a você determinar como estruturar, ler e escrever nesses campos, permitindo-lhe muita flexibilidade.

Para este exemplo, vamos usar um dado de credencial muito simples, contendo apenas a pergunta feita ao usuário:

```
{
 "pergunta":"aQuestion"
}
```

com um dado secreto igualmente simples, contendo apenas a resposta secreta:

```
{
 "resposta":"anAnswer"
}
```

Aqui a resposta será mantida em texto simples no banco de dados por uma questão de simplicidade, mas também seria possível ter um hash salgado para a resposta, como é o caso de senhas em Keycloak. Neste caso, os dados secretos também teriam que conter um campo para o sal, e as informações de dados de credenciais sobre o algoritmo, como o tipo de algoritmo usado e o número de iterações utilizadas. Para obter mais detalhes, consulte a implementação da classe `org.keycloak.models.credential.PasswordCredentialModel`.

No nosso caso, criamos a classe `SecretQuestionCredentialModel`:

```
classe pública SecretQuestionCredentialModel  estende  CredentialModel {
tipo de corda final estática pública = "SECRET_QUESTION";

   private private secretquestionCredentialData credencialData;
   secretcação secreta privadaSecretData secretaData;
```

Onde o `TYPE` é o credential_type que escrevemos no banco de dados. Para consistência, temos certeza de que esta String é sempre a referenciada ao obter o tipo para esta credencial. As classes `SecretQuestionCredentialData` e `SecretQuestionSecretData` são usadas para marechal e unmarshal the json:

```
classe pública SecretQuestionCredentialData  {

   pergunta final privada String;

   @JsonCreator
Public SecretQuestionCredentialData(@JsonProperty("pergunta") Pergunta de string) {
      esta.pergunta = pergunta;
   }

público String getQuestion() {
      pergunta de retorno;
   }
}

classe pública SecretQuestionSecretData  {

    resposta final privada string;

   @JsonCreator
SecretQuestionSecretData (@JsonProperty(resposta)Resposta de string) {
```

```
        esta.resposta = resposta;
    }

    cadeia pública getAnswer() {
        resposta de retorno;
    }
}
```

Para ser totalmente utilizável, os objetos SecretQuestionCredentialModel devem conter os dados de json brutos de sua classe pai, e os objetos não-marshalled em seus próprios atributos. Isso nos leva a criar um método que lê a partir de um modelo de credencial simples, como é criado ao ler a partir do banco de dados, para fazer um SecretQuestionCredentialModel:

```
SecretQuestionCredentialModel (SecretQuestionCredentialData credencialData,
SecretQuestionSecretData secretData) {
    esta.credencialData = credencialData;
    esta.secretData = secretData;
}

secretquestion públicaCredentialModel createFromCredentialModel
(CredencialModel(CredencialModel).{
    tente {
        SecretQuestionCredentialData credencialData = JsonSerialization.readValor
(credencialModel.getCredentialData(), SecretQuestionCredentialData.class);
        SecretQuestionSecretData secretData =
JsonSerialization.readValue(credencialModel.getSecretData(),
SecretQuestionSecretData.class);

SecretQuestionCredentialModel secretQuestionCredentialModel = novo
SecretQuestionCredentialModel (credencialData, secretData);
        secretQuestionCredentialModel.setUserLabel(credencialModel.getUserLabel());
        secretQuestionCredentialModel.setCreatedDate(credentialModel.getCreatedDate());
        secretQuestionCredentialModel.setType(TYPE);
        secretQuestionCredentialModel.setId(credencialModel.getId());
        secretQuestionCredentialModel.setSecretData(credentialModel.getSecretData());
        secretQuestionCredentialModel.setCredentialData(credentialModel.getCredentialData());
        retornar secretQuestionCredentialModel;
} catch  (IOException e){
        lançar novo RuntimeException(e);
    }
}
```

E um método para criar um SecretQuestionCredentialModel a partir da pergunta e resposta:

```
SecretQuestionCredentialModel (String question, String answer) {
credencialData = novo  SecretQuestionCredentialData (pergunta);
secretData = novo  SecretQuestionSecretData (resposta);
}
```

```
secretquestion público secretoCredentialModel createSecretQuestion (String question, String
answer) {
SecretQuestionCredentialModel = novo  SecretQuestionCredentialModel (pergunta, resposta);
   credencialModel.fillCredentialModelFields();
   devolução credencialModel;
}

preenchimento vazio privadoCredentialModelFields(){
   tente {
      setCredentialData(JsonSerialization.writeValueAsString(credencialData));
      setSecretData(JsonSerialization.writeValueAsString(secretData));
      setType (TYPE);
      setCreatedDate (Time.currentTimeMillis());
} catch  (IOException e) {
      lançar novo RuntimeException(e);
   }
}
```

Implementação de um Pronúncial Credencial

Como em todos os Provedores, para permitir que o Keycloak gere o CredencialProvider, exigimos uma CredencialProviderFactory. Para este requisito, criamos o SecretQuestionCredentialProviderFactory, cujo método `de criação` será chamado quando um SecretQuestionCredentialProvider for solicitado:

```
classe pública SecretQuestionCredentialProviderFactory  implementa
CredentialProviderFactory<SecretQuestionCredentialProvider> {

PROVIDER_ID final de corda estática pública =  "questão secreta";

   @Override
público String getId() {
      PROVIDER_ID de retorno;
   }

   @Override
credencial públicaProvider create (sessão KeycloakSession) {
      retornar novo SecretQuestionCredentialProvider(sessão);
   }
}
```

A interface CredencialProvider tem um parâmetro genérico que estende um Modelo de Credencial. No nosso caso, usamos o SecretQuestionCredentialModel que criamos:

```
classe pública SecretQuestionCredentialProvider  implementa
CredentialProvider<SecretQuestionCredentialModel>, CredentialInputValidator {
   logger final estático privado = Logger.getLogger (SecretQuestionCredentialProvider.class);

sessão de sessão de sessão de keycloaksession protegida;

secretquestionpedentialProvider (sessão KeycloakSession) {
```

```
        estasessão .session = sessão;
    }

UserCredentialStore privado getCredentialStore() {
        sessão de retorno.userCredentialManager();
    }
```

Também queremos implementar a interface CredentialInputValidator, pois isso permite que o Keycloak saiba que este provedor também pode ser usado para validar uma credencial para um Autenticador. Para a interface CredencialProvider, o primeiro método que precisa ser implementado é o método getType(). Isso simplesmente retornará a sequência de tipo 'SecretQuestionCredentialModel's TYPE:

```
@Override
cadeia pública getType() {
    retornar SecretQuestionCredentialModel.TYPE;
}
```

O segundo método é criar um SecretQuestionCredentialModel a partir de um Modelo de Credenciamento. Para este método, simplesmente chamamos o método estático existente de SecretQuestionCredentialModel:

```
@Override
público SecretQuestionCredentialModel getCredentialFromModel (credentialModel model) {
    retorno SecretQuestionCredentialModel.createFromCredentialModel (modelo);
}
```

Finalmente, temos os métodos para criar uma credencial e excluir uma credencial. Esses métodos chamam o userCredentialManagerdo KeycloakSession,que é responsável por saber onde ler ou escrever a credencial, por exemplo, armazenamento local ou armazenamento federado.

```
@Override
credencial públicaModel createCredential (RealmModel realm, UserModel user,
SecretQuestionCredentialModel credentialModel) {
se (credencialModel.getCreatedDate() == nulo){
        credentialModel.setCreatedDate (Time.currentTimeMillis());
    }
    retorno getCredentialStore().createCredential(reino, usuário, credenciamentoModel);
}

@Override
exclusão booleana públicaCredential (RealmModel realm, UserModel user, String credenciaid)
{
    retornar getCredentialStore().removeStoredCredential (reino, usuário, credencialId);
}
```

Para o CredentialInputValidator, o principal método a ser implementado é o isValid,que testa se uma credencial é válida para um determinado usuário em um determinado reino. Este é o método que é chamado pelo Autenticador quando ele busca validar a

entrada do usuário. Aqui, simplesmente precisamos verificar se a string de entrada é a registrada na Credencial:

```
@Override
público boolean isValid (RealmModel realm, UserModel user, CredentialInput input) {
se (!( instância de entrada do  UserCredentialModel)) {
      logger.debug("Instância esperada do UserCredentialModel for CredentialInput");
retorno falso;
   }
   se (!input.getType().equals(getType()) {
retorno falso;
   }
   Desafio de stringSPonse = input.getChallengeResponse();
se (desafioResponse ==  nulo){
retorno falso;
   }
   CredencialModel credencialModel = getCredentialStore().getStoredCredEntialById(reino,
usuário, input.getCredentialId());
   SecretQuestionCredentialModel sqcm = getCredentialFromModel (credencialModel);
   retorno sqcm.getSecretQuestionSecretData().getAnswer().equals(challengeResponse);
}
```

Os outros dois métodos a serem implementados são um teste se o CredencialProvider suporta o tipo de credencial dado e um teste para verificar se o tipo de credencial está configurado para um determinado usuário. Para o nosso caso, este último teste significa simplesmente verificar se o usuário tem uma credencial do tipo SECRET_QUESTION:

```
@Override
suportes booleanos públicosCredentialType (String credentialType) {
   retorno getType().equals(credencialType);
}

@Override
público boolean isConfiguredFor (RealmModel realm, UserModel user, String credentialType) {
se (!suportaCredentialType(credencialType))  retornar  falso;
   retorno !getCredentialStore().getStoredCredEntialsByType(reino, usuário,
credencialType).isEmpty();
}
```

Implementando um Autenticador

Ao implementar um autenticador que usa credenciais para autenticar um usuário, você deve ter o autenticador implementar a interface CredencialValidator. Essas interfaces têm uma classe que estende um Pronúncial como parâmetro e permitirão que o Keycloak chame diretamente os métodos do Pronúncial Credenciado. O único método que precisa ser implementado é o método `getCredentialProvider`, que em nosso exemplo permite que o SecretQuestionAuthenticator recupere o SecretQuestionCredentialProvider:

```
secretquestionpentialprovider público getCredentialProvider (KeycloakSession session) {
```

```
    retorno (SecretQuestionCredentialProvider)session.getProvider(CredencialProvider.class,
SecretQuestionCredentialProviderFactory.PROVIDER_ID);
}
```

Ao implementar a interface Autenticador, o primeiro método que precisa ser implementado é o método requerUser(). Por exemplo, este método deve retornar verdadeiro, pois precisamos validar a questão secreta associada ao usuário. Um provedor como kerberos retornaria falso deste método, pois pode resolver um usuário do cabeçalho de negociação. Este exemplo, no entanto, está validando uma credencial específica de um usuário específico.

O próximo método a ser implementado é o método configuradoFor(). Este método é responsável por determinar se o usuário está configurado para este autenticador em particular. No nosso caso, podemos apenas chamar o método implementado no SecretQuestionCredentialProvider

```
@Override
público boolean configuradoPara (Sessão KeycloakSession, ReinomModel, UsuárioModel
usuário) {
    retorno getCredentialProvider(session).isConfiguredFor(reino, usuário, getType(session));
}
```

O próximo método a ser implementado no Autenticador é o conjuntoRequiredActions(). Se configuradoPara() retornar falso e nosso autenticador de exemplo for necessário dentro do fluxo, este método será chamado, mas somente se o método isUserSetupAllowed do AuthenticatorFactory associado retornar verdadeiro. O método setRequiredActions() é responsável pelo registro de quaisquer ações necessárias que devem ser realizadas pelo usuário. Em nosso exemplo, precisamos registrar uma ação necessária que forçará o usuário a configurar a resposta para a pergunta secreta. Implementaremos este provedor de ação necessário mais tarde neste capítulo. Aqui está a implementação do método setRequiredActions().

```
    @Override
    conjunto de vazio públicoRequiredActions (KeycloakSession session, RealmModel realm,
UserModel user) {
        user.addRequiredAction("SECRET_QUESTION_CONFIG");
    }
```

Agora estamos entrando na carne da implementação do Autenticador. O próximo método a implementar é autenticar(). Este é o método inicial que o fluxo invoca quando a execução é visitada pela primeira vez. O que queremos é que se um usuário já respondeu a pergunta secreta na máquina do seu navegador, então o usuário não precisa responder a pergunta novamente, tornando essa máquina "confiável". O método autenticado() não é responsável pelo processamento da forma de pergunta secreta. Seu único propósito é renderizar a página ou continuar o fluxo.

```
@Override
autenticação do vazio público (AuthenticationFlowContext context) {
    se (hasCookie(contexto)) {
```

```
        contexto.sucesso();
retorno;
    }
    Desafio de resposta = context.form()
.createForm("segredo-questão.ftl");
    contexto.desafio(desafio);
}

booleano protegido hasCookie (AuthenticationFlowContext context) {
    Cookie cookie =
context.getHttpRequest().getHttpHeaders().getCookies().get().get("SECRET_QUESTION_ANSW
ERED");
resultado booleano = cookie !=  nulo;
    se (resultado) {
        System.out.println("Ignorando a pergunta secreta porque o cookie está definido");
    }
    resultado de retorno;
}
```

O método hasCookie() verifica se já existe um cookie definido no navegador, o que indica que a pergunta secreta já foi respondida. Se isso for real, marcamos o status desta execução como SUCESSO usando o método AuthenticationFlowContext.success() e retornando do método de autenticação().

Se o método hasCookie() retornar falso, devemos retornar uma resposta que torna a questão secreta html forma. AutenticaçãoFlowContext tem um método de formulário() que inicializa um construtor de páginas Freemarker com informações básicas apropriadas necessárias para construir o formulário. Este construtor de páginas se chama `org.keycloak.login.loginFormsProvider`. O método LoginFormsProvider.createForm() carrega um arquivo de modelo Freemarker a partir do seu tema de login. Além disso, você pode ligar para o método LoginFormsProvider.setAttribute() se quiser passar informações adicionais para o modelo Freemarker. Vamos passar por isso mais tarde.

Chamar loginFormsProvider.createForm() retorna um objeto de resposta JAX-RS. Em seguida, chamamos de AuthenticationFlowContext.challenge() passando nesta resposta. Isso define o status da execução como DESAFIO e, se a execução for necessária, este objeto JAX-RS Response será enviado para o navegador.

Assim, a página HTML pedindo a resposta para uma pergunta secreta é exibida para o usuário e o usuário entra na resposta e clica em enviar. A URL de ação do formulário HTML enviará uma solicitação HTTP para o fluxo. O fluxo acabará invocando o método de ação() da nossa implementação autenticador.

```
@Override
ação de vazio público (Contexto autenticaçãoFlowContext) {
    validado boolean = validarAnswer (contexto);
    se (!validado) {
        Desafio de resposta = context.form()
            .setError("badSecret")
```

```
.createForm("segredo-questão.ftl");
      contexto.failureChallenge (AuthenticationFlowError.INVALID_CREDENTIALS, desafio);
retorno;
   }
   setCookie (contexto);
   contexto.sucesso();
}
```

Se a resposta não for válida, reconstruíremos o Formulário HTML com uma mensagem de erro adicional. Em seguida, chamamos de AuthenticationFlowContext.failureChallenge() passando a razão do valor e da resposta JAX-RS. failureChallenge() funciona da mesma forma que o Desafio(), mas também registra a falha para que possa ser analisada por qualquer serviço de detecção de ataque.

Se a validação for bem sucedida, então definimos um cookie para lembrar que a pergunta secreta foi respondida e chamamos de AuthenticationFlowContext.success().

A validação em si obtém os dados recebidos do formulário e chama o método isValid do SecretQuestionCredentialProvider. Você vai notar que há uma seção do código sobre obter a credencial. Isso porque se o Keycloak estiver configurado para permitir vários tipos de autenticadores alternativos, ou se o usuário poderia gravar várias credenciais do tipo SECRET_QUESTION (por exemplo, se permitirmos escolher entre várias perguntas, e permitimos que o usuário tivesse respostas para mais de uma dessas perguntas), então o Keycloak precisa saber qual credencial está sendo usada para registrar o usuário. Caso haja mais de uma credencial, o Keycloak permite que o usuário escolha durante o login qual credencial está sendo usada, e as informações são transmitidas pelo formulário para o Autenticador. Caso o formulário não apresente essas informações, o id de credencial utilizado é dado pelo método `de credencial getDefaultCredential` do Credencial, que retornará a credencial "mais preferida" do tipo correto do usuário,

```
validação booleana protegidaAnswer (AutenticaçãoFlowContext contexto) {
   MultivalorizadoMap<String, String> formData =
context.getHttpRequest().getDecodedFormParameters();
   Segredo de string = formData.getFirst("secret_answer");
   Credencial de string = formData.getFirst("credencialId");
se (credencial ==  nulo || credencialId.isEmpty()) {
      credencialId = getCredentialProvider (context.getSession())
            .getDefaultCredential (context.getSession(), context.getRealm(),
context.getUser()).getId();
   }

Entrada UserCredentialModel = novo  UserCredentialModel (credencialId, getType
(context.getSession()), secreto);
   retorno getCredentialProvider (context.getSession()).isValid (context.getRealm(),
context.getUser(), entrada);
}
```

O próximo método é o setCookie(). Este é um exemplo de fornecimento de configuração para o Autenticador. Neste caso, queremos que a idade máxima do biscoito seja configurável.

```
conjunto de vazio protegidoCookie (AuthenticationFlowContext context) {
   AuthenticatorConfigModel config = context.getAuthenticatorConfig();
int maxCookieAge =  60  *  60  *  24  *  30; // 30 dias
se (config !=  nulo){
      maxCookieAge = Integer.valueOf(config.getConfig().get("cookie.max.age"));

   }
URI uri =
context.getUriInfo().getBaseUriBuilder().path().path(context.getRealm().getName()).build();
addCookie (contexto, "SECRET_QUESTION_ANSWERED",  "verdadeiro",
         uri.getRawPath(),
nulo,  nulo,
         maxCookieAge,
falso,  verdadeiro);
}
```

Obtemos um AuthenticatorConfigModel do método AuthenticationFlowContext.getAuthenticatorConfig(). Se existe configuração, tiramos a idade máxima de confiança dela. Veremos como podemos definir o que deve ser configurado quando falamos sobre a implementação do AuthenticatorFactory. Os valores de config podem ser definidos dentro do console de administração se você configurar definições de config na implementação do AuthenticatorFactory.

```
@Override
credencial públicaTypeMetadata getCredentialTypeMetadata() {
      retornar CredencialTypeMetadata.builder()
            .type(getType())
            .categoria (CredentialTypeMetadata.Category.TWO_FACTOR)
            .displayName (SecretQuestionCredentialProviderFactory.PROVIDER_ID)
.helpText("texto-pergunta-questão secreta")
            .createAction(SecretQuestionAuthenticatorFactory.PROVIDER_ID)
.removível(falso)
            .build(sessão);
   }
```

O último método na classe SecretQuestionCredentialProvider é getCredentialTypeMetadata(), que é um método abstrato da interface CredencialProvider. Cada provedor de Credencial tem que fornecer e implementar este método. O método retorna uma instância de CredencialTypeMetadata, que deve pelo menos incluir tipo e categoria de autenticador, displayName e item removível. Neste exemplo, o construtor pega o tipo de autenticador do método getType(), categoria é Two Factor (o autenticador pode ser usado como segundo fator de autenticação) e removível, que é configurado como falso (o usuário não pode remover algumas credenciais previamente registradas).

Outros itens do construtor são o helpText (será mostrado ao usuário em várias telas), criarAction (o provedorID da ação necessária, que pode ser usado pelo usuário para criar nova credencial) ou atualizarAction (mesmo que criarAction, mas em vez de criar a nova credencial, ele atualizará a credencial).

Implementando um AuthenticatorFactory

O próximo passo nesse processo é implementar uma AuthenticatorFactory. Esta fábrica é responsável por instanciar um Autenticador. Ele também fornece metadados de implantação e configuração sobre o Autenticador.

O método getId() é apenas o nome único do componente. O método create() é chamado pelo tempo de execução para alocar e processar o Autenticador.

```
classe pública SecretQuestionAuthenticatorFactory  implementa  AuthenticatorFactory,
ConfigurávelAuthenticatorFactory {

PROVIDER_ID final de string estática pública =  "secret-question-authenticator";
secretquestion final privadoAuthenticator SINGLETON =  novo  SecretQuestionAuthenticator();

   @Override
público String getId() {
      PROVIDER_ID de retorno;
   }

   @Override
   criação de autenticador público (sessão KeycloakSession) {
      retorno SINGLETON;
   }
```

A próxima coisa pela qual a fábrica é responsável é especificar os switches de exigência permitidos. Embora existam quatro tipos de requisitos diferentes: IMPLEMENTAções ALTERNATIVAS, NECESSÁRIAS, CONDICIONAIS, INCAPACITADAS, AutenticadoresFactory podem limitar quais opções de exigência são mostradas no console administrativo ao definir um fluxo. O CONDITIONAL só deve ser sempre usado para subfluxos, e a menos que haja uma boa razão para fazer o contrário, a exigência de um autenticador deve ser NECESSÁRIA, ALTERNATIVA e DEFICIENTE:

```
autenticação estática privadaExecutionModel.Requirement[]  REQUIREMENT_CHOICES = {
         AutenticaçãoExecutionModel.Requirement.REQUIRED,
         AutenticaçãoExecutionModel.Requirement.ALTERNATIVA,
         AutenticaçãoExecutionModel.Requirement.DISABLED
   };
   @Override
Autenticação públicaExecutionModel.Requirement[]  getRequirementChoices() {
      REQUIREMENT_CHOICES de retorno;
   }
```

O AuthenticatorFactory.isUserSetupAllowed() é um sinalizador que informa ao gerenciador de fluxo se o método Authenticator.setRequiredActions(). Se um Autenticador não estiver configurado para um usuário, o gerenciador de fluxo será UserSetupAllowed(). Se for falso, então o fluxo aborta com um erro. Se ele retornar verdadeiro, então o gerenciador de fluxo invocará Authenticator.setRequiredActions().

```
   @Override
   público booleano isUserSetupAllowed() {
retorno verdadeiro;
   }
```

Os próximos métodos definem como o Autenticador pode ser configurado. O método isConfigurable() é um sinalizador que especifica ao console administrativo se o Autenticador pode ser configurado dentro de um fluxo. O método getConfigProperties() retorna uma lista de objetos ProviderConfigProperty. Esses objetos definem um atributo de configuração específico.

```
   @Override
Lista pública<ProviderConfigProperty> getConfigProperties() {
configproperties de retorno;
   }

Lista final estática privada<ProviderConfigProperty> configProperties =  novo
ArrayList<ProviderConfigProperty>();

   estática {
      Propriedade Do ProvedorConfigProperty;
propriedade = novo  ProvedorConfigProperty();
      property.setName("cookie.max.age");
      property.setLabel("Cookie Max Age");
      property.setType (ProviderConfigProperty.STRING_TYPE);
property.setHelpText("Max age em segundos do SECRET_QUESTION_COOKIE." );
      configProperties.add(propriedade);
   }
```

Cada ProvedorConfigProperty define o nome da propriedade config. Esta é a chave usada no mapa de config armazenado no AuthenticatorConfigModel. O rótulo define como a opção de config será exibida no console administrativo. O tipo define se é um string, booleano ou outro tipo. O console administrativo exibirá diferentes entradas de interface do usuário, dependendo do tipo. O texto de ajuda é o que será mostrado na dica de ferramenta para o atributo config no console administrativo. Leia o javadoc do ProviderConfigProperty para obter mais detalhes.

O resto dos métodos são para o console administrativo. getHelpText() é o texto da dica de ferramenta que será mostrado quando você estiver escolhendo o Autenticador que deseja vincular a uma execução. getDisplayType() é o texto que será mostrado no console administrativo ao listar o Autenticador. getReferenceCategory() é apenas uma categoria a que o Autenticador pertence.

Adicionando formulário autenticador

Keycloak vem com um tema Freemarker e um mecanismo de modelo. O método createForm() que você chamou dentro do autenticador() da sua classe Autenticador, constrói uma página HTML a partir de um arquivo dentro do seu tema de login: secret-question.ftl. Este arquivo deve ser adicionado aos recursos/modelos temáticos em seu JAR, consulte o Provedor de Recursos Temáticos para obter mais detalhes.

Vamos dar uma olhada maior na questão secreta.ftl Aqui está um pequeno trecho de código:

```
<form id=classe "kc-totp-login-form" ="${properties.kcFormClass!}" ação=método
"${url.loginAction}" ="post">
<div class ="${properties.kcFormGroupClass!}" >
< classediv ="${properties.kcLabelWrapperClass!}" >
< rótulo para=classe "totp"
="${properties.kcLabelClass!}" >${msg("loginSecretQuestion"}</label>
            </div>

<div class ="${properties.kcInputWrapperClass!}" >
< id deentrada= nome "totp" =tipo "secret_answer" = classe"texto"
="${propriedades.kcInputClass!}" />
            </div>
          </div>
      </formulário>
```

Qualquer pedaço de texto incluído em ${} corresponde a um atributo ou função de modelo. Se você ver a ação do formulário, verá que ele aponta para ${url.loginAction}. Esse valor é gerado automaticamente quando você invoca o método AuthenticationFlowContext.form(). Você também pode obter esse valor chamando o método AuthenticationFlowContext.getActionURL() no código Java.

Você também verá ${properties.someValue}. Essas correspondem a propriedades definidas em seu tema.properties arquivo do nosso tema. ${msg("someValue")} corresponde aos pacotes de mensagens internacionalizados (.arquivos de propriedades) incluídos com as mensagens/diretórios do tema de login. Se você estiver apenas usando inglês, basta adicionar o valor do loginSecretQuestion. Esta deve ser a pergunta que você deseja fazer ao usuário.

Quando você chama de AuthenticationFlowContext.form() isso lhe dá uma instância LoginFormsProvider. Se você ligasse, LoginFormsProvider.setAttribute("foo", "bar"), o valor de "foo" estaria disponível para referência em seu formulário como ${foo}. O valor de um atributo pode ser qualquer feijão Java também.

Se você olhar para a parte superior do arquivo, verá que estamos importando um modelo:

```
< #importar "select.ftl" como layout>
```

Importar este modelo, em vez do modelo padrão.ftl permite que o Keycloak exiba uma caixa suspensa que permite ao usuário selecionar uma credencial ou execução diferente.

Adicionando autenticador a um fluxo

Adicionar um Autenticador a um fluxo deve ser feito no console administrativo. Se você for ao item do menu Autenticação e for para a guia Fluxo, poderá visualizar os fluxos definidos no momento. Você não pode modificar fluxos incorporados, então, para adicionar o Autenticador que criamos, você tem que copiar um fluxo existente ou criar o seu próprio. Nossa esperança é que a interface do usuário seja suficientemente clara para que você possa determinar como criar um fluxo e adicionar o Autenticador. Para obter mais detalhes, consulte o capítulo Fluxos de Autenticação no Guia de Administração do Servidor .

Depois de criar seu fluxo, você tem que vinculá-lo à ação de login a que deseja vinculá-lo. Se você for ao menu Autenticação e for para a guia Vinculações, você verá opções para vincular um fluxo ao navegador, registro ou fluxo de subvenção direta.

Passo a passo de ação necessário

Nesta seção discutiremos como definir uma ação necessária. Na seção Autenticador, você pode ter se perguntado: "Como obteremos a resposta do usuário para a pergunta secreta inserida no sistema?". Como mostramos no exemplo, se a resposta não for configurada, uma ação necessária será acionada. Esta seção discute como implementar a ação necessária para o Autenticador de Perguntas Secretas.

Aulas e implantação de embalagens

Você empacotará suas aulas dentro de um único frasco. Este frasco não precisa ser separado de outras classes de provedores, mas deve conter um arquivo chamado org.keycloak.authentication.RequiredActionFactory e deve ser contido no META-INF/serviços/ diretório do seu frasco. Este arquivo deve listar o nome de classe totalmente qualificado de cada implementação RequiredActionFactory que você tem no frasco. Por exemplo:

```
org.keycloak.examples.authenticator.SecretQuestionRequiredActionFactory
```

Este serviço/arquivo é usado pelo Keycloak para digitalizar os provedores que ele tem que carregar no sistema.

Para implantar este frasco, basta copiá-lo para o diretório autônomo/implantações.

Implementar oProvider deAction Obrigatório

As ações necessárias devem primeiro implementar a interface RequiredActionProvider. O RequiredActionProvider.requiredActionChallenge() é a chamada inicial do gerenciador

de fluxo para a ação necessária. Este método é responsável por renderizar o formulário HTML que conduzirá a ação necessária.

```
  @Override
  vazio público requerActionChallenge (Contexto RequiredActionContext) {
     Desafio de resposta = context.form().createForm("secret_question_config.ftl");
     contexto.desafio(desafio);

  }
```

Você vê que o RequiredActionContext tem métodos semelhantes ao AuthenticationFlowContext. O método form() permite que você torne a página a partir de um modelo Freemarker. A URL de ação é predefinida pela chamada para este método(). Você só precisa referenciá-lo dentro do seu formulário HTML. Vou te mostrar isso mais tarde.

O método de desafio() notifica o gerenciador de fluxos de que uma ação necessária deve ser executada.

O próximo método é responsável pelo processamento da entrada a partir da forma HTML da ação necessária. A URL de ação do formulário será roteado para o método RequiredActionProvider.processAction()

```
  @Override
  processo de vazio públicoAction (RequiredActionContext context) {
Resposta de string =
(context.getHttpRequest().getDecodedFormParameters().getFirst("resposta"));
Modelo UserCredentialValueModel = novo  UserCredentialValueModel();
      model.setValor (resposta);
      model.setType (SecretQuestionAuthenticator.CREDENTIAL_TYPE);
      context.getUser().updateCredentialDirect(modelo);
      contexto.sucesso();
  }
```

A resposta é retirada do posto de formulário. Um UserCredentialValueModel é criado e o tipo e o valor da credencial são definidos. Em seguida, o UserModel.updateCredentialDirectly() é invocado. Finalmente, RequiredActionContext.success() notifica o contêiner de que a ação necessária foi bem sucedida.

Implementar o RequiredActionFactory

Esta aula é muito simples. É apenas responsável pela criação da instância do provedor de ação necessária.

```
SecretQuestion secretquestionRequiredActionFactory  implementa  RequiredActionFactory {

secretquestion final estático privadoRequiredAction SINGLETON =  nova
SecretQuestionRequiredAction();
```

```
   @Override
   público RequiredActionProvider create (Sessão KeycloakSession) {
      retorno SINGLETON;
   }


   @Override
público String getId() {
      SecretQuestionRequiredAction.PROVIDER_ID de retorno;
   }

   @Override
   cadeia pública getDisplayText() {
retornar "Questão Secreta";
   }
```

O método getDisplayText() é apenas para o console administrativo quando ele quer exibir um nome amigável para a ação necessária.

Habilitar ação necessária

A última coisa que você tem que fazer é ir para o console administrativo. Clique no menu esquerdo de Autenticação. Clique na guia Ações Necessárias. Clique no botão Registrar e escolha sua nova Ação Necessária. Sua nova ação necessária deve agora ser exibida e habilitada na lista de ações necessárias.

Modificando/estendendo o formulário de inscrição

É inteiramente possível que você implemente seu próprio fluxo com um conjunto de Autenticadores para alterar totalmente a forma como o registro é feito no Keycloak. Mas o que você geralmente vai querer fazer é apenas adicionar um pouco de validação para a página de registro fora da caixa. Um SPI adicional foi criado para poder fazer isso. Ele basicamente permite adicionar validação de elementos de formulário na página, bem como inicializar atributos e dados do UserModel após o registro do usuário. Analisaremos tanto a implementação do processamento de registro de perfil do usuário quanto o plugin do Google Recaptcha.

Interface de ação de formulário de implementação

A interface principal que você tem que implementar é a interface FormAction. A FormAction é responsável pela renderização e processamento de uma parte da página. A renderização é feita no método buildPage(), a validação é feita no método valid () validação, as operações de validação pós-validação são feitas com sucesso(). Vamos primeiro dar uma olhada no método buildPage() do plugin Recaptcha.

```
   @Override
   buildPage de vazio público (formcontext context, loginFormsProvider form) {
```

```
      AuthenticatorConfigModel captchaConfig = context.getAuthenticatorConfig();
se (captchaConfig ==  nulo || captchaConfig.getConfig() ==  nulo
            || captchaConfig.getConfig().get(SITE_KEY) == nulo
            || captchaConfig.getConfig().get(SITE_SECRET) == nulo
            ) {
form.addError(novo  FormMessage(nulo,Messages.RECAPTCHA_NOT_CONFIGURED));
retorno;
      }
      Site de stringKey = captchaConfig.getConfig().get(SITE_KEY);
form.setAttribute("recaptchaRequired",  true);
      form.setAttribute("recaptchaSiteKey", siteKey);
form.addScript("https://www.google.com/recaptcha/api.js");
   }
```

O método Recaptcha buildPage() é um retorno de chamada pelo fluxo de formulário para ajudar a renderizar a página. Ele recebe um parâmetro de formulário que é um LoginFormsProvider. Você pode adicionar atributos adicionais ao provedor de formulários para que eles possam ser exibidos na página HTML gerada pelo modelo freemarker de registro.

O código acima é do plugin de recaptcha de registro. A recaptcha requer algumas configurações específicas que devem ser obtidas a partir da configuração. As Ações de Formulários são configuradas exatamente da mesma forma que os Autenticadores são. Neste exemplo, puxamos a chave do site do Google Recaptcha da configuração e adicionamos-a como um atributo ao provedor de formulários. Nosso arquivo de modelo de registro pode ler este atributo agora.

Recaptcha também tem o requisito de carregar um script JavaScript. Você pode fazer isso chamando loginFormsProvider.addScript() passando na URL.

Para o processamento do perfil do usuário, não há informações adicionais que ele precise adicionar ao formulário, de modo que seu método buildPage() esteja vazio.

A próxima parte carnínua desta interface é o método de validação(). Isso é chamado imediatamente após receber um post de formulário. Vamos ver o plugin da Recaptcha primeiro.

```
   @Override
   validação do vazio público (validação Contexto de contexto) {
      MultivalorizadoMap<String, String> formData =
context.getHttpRequest().getDecodedFormParameters();
Erros <formmessage> = novo  ArrayList<>();
sucesso booleano =  falso;

      Captcha de cordas = formData.getFirst(G_RECAPTCHA_RESPONSE);
      se (! Validation.isBlank(captcha)) {
         AuthenticatorConfigModel captchaConfig = context.getAuthenticatorConfig();
         Segredo de string = captchaConfig.getConfig().get(SITE_SECRET);

         sucesso = validarRecaptcha (contexto, sucesso, captcha, segredo);
```

```
      }
      se (sucesso) {
         contexto.sucesso();
} outra coisa  {
erros.add(novo  FormMessage(nulo,Messages.RECAPTCHA_FAILED));
         formData.remove(G_RECAPTCHA_RESPONSE);
         context.validationError (formData, erros);
retorno;


      }
   }
```

Aqui obtemos os dados de formulário que o widget Recaptcha adiciona ao formulário. Obtemos a chave secreta Recaptcha da configuração. Então validamos a recaptcha. Se for bem-sucedido, validationContext.success() é chamado. Se não, invocamos validationContext.validationError() passando no formulárioData (para que o usuário não precise reinserir dados), também especificamos uma mensagem de erro que queremos exibida. A mensagem de erro deve apontar para uma propriedade de pacote de mensagens nos pacotes de mensagens internacionalizados. Para outras extensões de registro validar() pode ser validar o formato de um elemento de formulário, ou seja, um atributo de e-mail alternativo.

Vamos também olhar para o plugin do perfil do usuário que é usado para validar endereço de e-mail e outras informações do usuário ao se registrar.

```
   @Override
   validação do vazio público (validação Contexto de contexto) {
      MultivalorizadoMap<String, String> formData =
context.getHttpRequest().getDecodedFormParameters();
Erros <formmessage> = novo  ArrayList<>();

      Evento de cordasError = Errors.INVALID_REGISTRATION;

se (Validation.isBlank(formData.getFirst((RegistrationPage.FIELD_FIRST_NAME))) {
erros.add(novo  FormMessage(RegistrationPage.FIELD_FIRST_NAME,
Messages.MISSING_FIRST_NAME));
      }

se (Validation.isBlank(formData.getFirst((RegistrationPage.FIELD_LAST_NAME))) {
errors.add(novo  FormMessage(RegistrationPage.FIELD_LAST_NAME,
Messages.MISSING_LAST_NAME));
      }

      E-mail de string = formData.getFirst(Validation.FIELD_EMAIL);
      se (Validation.isBlank(email)) {
erros.add(novo  FormMessage(RegistrationPage.FIELD_EMAIL, Messages.MISSING_EMAIL));
} mais se  (! Validation.isEmailValid(email)) {
         formData.remove(Validation.FIELD_EMAIL);
erros.add(novo  FormMessage(RegistrationPage.FIELD_EMAIL, Messages.INVALID_EMAIL));
      }
```

```
se (context.getSession().users().getUserByEmail (e-mail, context.getRealm()) != nulo){
        formData.remove(Validation.FIELD_EMAIL);
errors.add(novo FormMessage(RegistrationPage.FIELD_EMAIL, Messages.EMAIL_EXISTS));
    }

se (erros.tamanho() > 0) {
        context.validationError (formData, erros);
retorno;

} outra coisa {
        contexto.sucesso();
    }
  }
```

Como você pode ver, este método de validação () de processamento de perfil do usuário garante que o e-mail, primeiro e sobrenome sejam preenchidos no formulário. Ele também garante que o e-mail esteja no formato certo. Se alguma dessas validações falhar, uma mensagem de erro será feita na fila para renderização. Todos os campos de erro são removidos dos dados do formulário. As mensagens de erro são representadas pela classe FormMessage. O primeiro parâmetro do construtor desta classe toma o id elemento HTML. A entrada por erro será destacada quando o formulário for re renderizado. O segundo parâmetro é um id de referência de mensagem. Este id deve corresponder a uma propriedade em um dos arquivos localizados do pacote de mensagens. no tema.

Depois de todas as validações terem sido processadas, o fluxo de formulários então invoca o método FormAction.success(). Para recaptcha isso é um não-op, então não vamos passar por cima dele. Para o processamento do perfil do usuário, este método preenche valores no usuário cadastrado.

```
  @Override
  sucesso do vazio público (contexto FormContext) {
    Usuário do UserModel = context.getUser();
    MultivalorizadoMap<String, String> formData =
context.getHttpRequest().getDecodedFormParameters();
    user.setFirstName(formData.getFirst(RegistrationPage.FIELD_FIRST_NAME));
    user.setLastName(formData.getFirst(RegistrationPage.FIELD_LAST_NAME));
    user.setEmail(formData.getFirst(RegistrationPage.FIELD_EMAIL));
  }
```

Implementação bastante simples. O UserModel do usuário recém-registrado é obtido do FormContext. Os métodos apropriados são chamados para inicializar os dados do UserModel.

Finalmente, você também é obrigado a definir uma classe FormActionFactory. Esta classe é implementada de forma semelhante à AuthenticatorFactory, então não vamos passar por cima dela.

Embalando a Ação

Você empacotará suas aulas dentro de um único frasco. Este frasco deve conter um arquivo chamado org.keycloak.authentication.FormActionFactory e deve ser contido no META-INF/serviços/ diretório do seu frasco. Este arquivo deve listar o nome de classe totalmente qualificado de cada implementação FormActionFactory que você tem no frasco. Por exemplo:

```
org.keycloak.authentication.forms.RegistrationProfile
org.keycloak.authentication.forms.RegistrationRecaptcha
```

Este serviço/arquivo é usado pelo Keycloak para digitalizar os provedores que ele tem que carregar no sistema.

Para implantar este frasco, basta copiá-lo para o diretório autônomo/implantações.

Adicionando FormAction ao fluxo de registro

A adição de um FormAction a um fluxo de página de registro deve ser feita no console administrativo. Se você for ao item do menu Autenticação e for para a guia Fluxo, poderá visualizar os fluxos definidos no momento. Você não pode modificar fluxos incorporados, então, para adicionar o Autenticador que criamos, você tem que copiar um fluxo existente ou criar o seu próprio. Espero que a interface do usuário seja intuitiva o suficiente para que você possa descobrir por si mesmo como criar um fluxo e adicionar o FormAction.

Basicamente você terá que copiar o fluxo de registro. Em seguida, clique no menu Ações à direita do Formulário de Inscrição e escolha "Adicionar execução" para adicionar uma nova execução. Você escolherá o FormAction na lista de seleção. Certifique-se de que seu FormAction vem após "Criação de usuário de registro" usando os botões de baixa para movê-lo se o FormAction ainda não estiver listado após "Criação de Usuário de Registro". Você deseja que seu FormAction venha após a criação do usuário porque o método de sucesso () de Criação de Usuários de Registro é responsável pela criação do novo UserModel.

Depois de criar seu fluxo, você tem que vinculá-lo ao registro. Se você for ao menu Autenticação e for para a guia Vinculações, você verá opções para vincular um fluxo ao navegador, registro ou fluxo de subvenção direta.

Modificando o fluxo de senha/credencial esquecido

Keycloak também tem um fluxo de autenticação específico para senha esquecida, ou melhor, redefinição de credencial iniciada por um usuário. Se você for para a página de fluxos de fluxos do console administrativo, haverá um fluxo de "credenciais de reset". Por padrão, keycloak pede o e-mail ou nome de usuário do usuário e envia um e-mail

para eles. Se o usuário clicar no link, então ele poderá redefinir sua senha e OTP (se um OTP tiver sido configurado). Você pode desativar a redefinição automática de OTP desativando o autenticador "Reset OTP" no fluxo.

Você também pode adicionar funcionalidade adicional a esse fluxo. Por exemplo, muitas implantações gostariam que o usuário respondesse a uma ou mais perguntas secretas adicionais ao envio de um e-mail com um link. Você pode expandir o exemplo de pergunta secreta que vem com o distro e incorporá-lo no fluxo de credencial de reset.

Uma coisa a notar se você está estendendo o fluxo de credenciais de reset. O primeiro "autenticador" é apenas uma página para obter o nome de usuário ou e-mail. Se o nome de usuário ou e-mail existir, então o AuthenticationFlowContext.getUser() retornará o usuário localizado. Caso contrário, isso será nulo. Este formulário **NÃO solicitará** ao usuário para inserir em um e-mail ou nome de usuário se o e-mail ou nome de usuário anterior não existisse. Você precisa evitar que os atacantes sejam capazes de adivinhar usuários válidos. Assim, se o AuthenticationFlowContext.getUser() retornar nulo, você deve prosseguir com o fluxo para fazer parecer que um usuário válido foi selecionado. Sugiro que se você quiser adicionar perguntas secretas a este fluxo, você deve fazer essas perguntas depois que o e-mail é enviado. Em outras palavras, adicione seu autenticador personalizado após o autenticador "Enviar e-mail de reset".

Modificando o fluxo de login do primeiro corretor

O primeiro fluxo de login do corretor é usado durante o primeiro login com algum provedor de identidade. O Termo Primeiro Login significa que ainda não existe a conta Keycloak vinculada à conta do provedor de identidade autenticada em particular. Para obter mais detalhes sobre esse fluxo, consulte o capítulo De Intermediação de identidade no Guia de Administração do Servidor .

Autenticação de clientes

O Keycloak realmente suporta autenticação plugável para aplicativos clientes OpenID Connect. A autenticação do cliente (aplicativo) é usada sob o capô pelo adaptador Keycloak durante o envio de quaisquer solicitações de backchannel para o servidor Keycloak (como a solicitação de código de troca para acessar token após autenticação bem sucedida ou solicitação de atualização do token). Mas a autenticação do cliente também pode ser usada diretamente por você durante subvenções de acesso direto (representadas pelo OAuth2 Resource Owner Password Credentials Flow)ou durante a autenticação da conta de serviço (representada pelo OAuth2 Client Credentials Flow).

Para obter mais detalhes sobre o adaptador Keycloak e os fluxos OAuth2, consulte o Guia de Aplicações e Serviços de Proteção.

Implementações padrão

Na verdade, o Keycloak tem 2 implementações padrão de autenticação do cliente:

Autenticação tradicional com client_id e client_secret

Este é o mecanismo padrão mencionado na especificação OpenID Connect ou OAuth2 e o Keycloak o suporta desde os primeiros dias. O cliente público precisa incluir client_id parâmetro com seu ID na solicitação POST (por isso não é de fato não autenticado) e o cliente confidencial precisa incluir Autorização: Cabeçalho básico com o clienteId e clienteSecret usado como nome de usuário e senha.

Autenticação com JWT assinado

Isso é baseado nos perfis de token do portador JWT para especificação OAuth 2.0. O cliente/adaptador gera o JWT e assina-o com sua chave privada. O Keycloak então verifica o JWT assinado com a chave pública do cliente e autentica o cliente com base nele.

Veja o exemplo de demonstração e especialmente os exemplos/pré-configurado-demo/aplicativo de produto para o aplicativo de exemplo mostrando o aplicativo usando autenticação do cliente com JWT assinado.

Implemente seu próprio autenticador de clientes

Para conectar seu próprio autenticador cliente, você precisa implementar poucas interfaces tanto no lado do cliente (adaptador) quanto no servidor.

Lado do cliente

Aqui você precisa implementar org.keycloak.adapters.authentication.ClientCredentialsProvider e colocar a implementação para:

- seu arquivo WAR em WEB-INF/classes . Mas, neste caso, a implementação pode ser usada apenas para este único aplicativo WAR
- Algum arquivo JAR, que será adicionado no WEB-INF/lib da sua GUERRA
- Alguns arquivos JAR, que serão usados como módulo jboss e configurados na estrutura de implantação jboss.xml do seu WAR. Em todos os casos, você também precisa criar o arquivo META-INF/services/org.keycloak.adapters.authentication.ClientCredentialsProvider na GUERRA ou no seu JAR.

Lado do servidor

Aqui você precisa implementar org.keycloak.authentication.ClientAuthenticatorFactory e org.keycloak.authentication.ClientAuthenticator . Você também precisa adicionar o arquivo META-INF/services/org.keycloak.authentication.ClientAuthenticatorFactory com o nome

das classes de implementação. Consulte autenticadores para obter mais detalhes.

SPI de token de ação

Um token de ação é uma instância especial do Json Web Token (JWT) que permite ao seu portador realizar algumas ações, por exemplo, redefinir uma senha ou validar endereço de e-mail. Eles geralmente são enviados aos usuários na forma de um link que aponta para um endpoint processando tokens de ação para um determinado reino.

O Keycloak oferece quatro tipos básicos de tokens que permitem ao portador:

- Redefinir credenciais
- Confirmar endereço de e-mail
- Executar as ações necessárias
- Confirmar a vinculação de uma conta com conta no provedor de identidade externa

Além disso, é possível implementar qualquer funcionalidade que inicie ou modifique a sessão de autenticação usando o SPI do token de ação, detalhes dos quais estão descritos no texto abaixo.

Anatomia do Token de Ação

O token de ação é um Token Web Json padrão assinado com a chave de reino ativo onde a carga contém vários campos:

- `typ` - Identificação da ação (por exemplo, `verificar-e-mail`)
- `iat` e `exp` - Tempos de validade de token
- `sub` - ID do usuário
- `azp` - Nome do cliente
- `iss` - Emissor - URL do reino emissor
- `aud` - Audiência - lista contendo URL do reino emissor
- `asid` - ID da sessão de autenticação*(opcional)*
- `nonce` - Nonce aleatório para garantir a exclusividade de uso se a operação só pode ser executada uma vez*(opcional*)

Além disso, um token de ação pode conter qualquer número de campos personalizados serializáveis no JSON.

Processamento de tokens de ação

Quando um token de ação é passado para um ponto final keycloak *KEYCLOAK_ROOT*/auth/realms/master/login-actions/action-token via parâmetro-chave, ele é validado e um manipulador de token de ação adequado é executado. **O processamento sempre ocorre em um contexto de uma sessão de autenticação,** seja uma nova ou o serviço de token de ação se junta a uma sessão de autenticação existente (detalhes são descritos abaixo). O manipulador de tokens de ação pode executar ações prescritas pelo token (muitas vezes altera a sessão de autenticação) e resulta em uma resposta HTTP (por exemplo, pode continuar na autenticação ou exibir uma página de informações/erro). Estas etapas estão detalhadas abaixo.

1. **Validação básica do token de ação.** A validade da assinatura e do tempo é verificada e o manipulador de tokens de ação é determinado com base no campo de digitação.
2. **Determinando a sessão de autenticação.** Se a URL do token de ação for aberta no navegador com a sessão de autenticação existente, e o token contiver o ID da sessão de autenticação correspondente à sessão de autenticação do navegador, a validação e o manuseio do token de ação anexarão esta sessão de autenticação em andamento. Caso contrário, o manipulador de tokens de ação cria uma nova sessão de autenticação que substitui qualquer outra sessão de autenticação presente naquele momento no navegador.
3. **Validações de tokens específicas para tipo de token.** A lógica do ponto final do token de ação valida que o usuário(subcampo) e o cliente(azp) do token existem, são válidos e não desabilitados. Em seguida, valida todas as validações personalizadas definidas no manipulador de tokens de ação. Além disso, o manipulador de tokens pode solicitar que este token seja de uso único. Tokens já usados seriam então rejeitados pela lógica de ponto final do token de ação.
4. **Executando a ação.** Após todas essas validações, o código do manipulador de tokens de ação é chamado de que executa a ação real de acordo com os parâmetros no token.
5. **Invalidação de tokens de uso único.** Se o token for definido como uso único, uma vez que o fluxo de autenticação termine, o token de ação será invalidado.

Implemente seu próprio token de ação e seu manipulador

Como criar um token de ação

Como o token de ação é apenas um JWT assinado com poucos campos obrigatórios (ver Anatomia do token de ação acima), ele pode ser serializado e assinado como tal usando a classe JWSBuilder da Keycloak. Desta forma já foi implementado no método serialize (session, realm, uriInfo) de org.keycloak.authentication.actiontoken.DefaultActionToken e pode ser aproveitado pelos implementadores usando essa classe para tokens em vez de JsonWebTokensimples.

O exemplo a seguir mostra a implementação de um token de ação simples. Note que a classe deve ter um construtor privado sem quaisquer argumentos. Isso é necessário para deserializar a classe token da JWT.

```
import org.keycloak.authentication.actiontoken.DefaultActionToken;

demoActionToken de classe pública estende DefaultActionToken {

TOKEN_TYPE final de string estática pública = "my-demo-token";

demoActionToken público (Usuário de stringId, int absoluteExpirationInSecs, String
compoundAuthenticationSessionId) {
super(userId, TOKEN_TYPE, absoluteExpirationInSecs,
nulo,compostoAuthenticationSessionId);
   }

demoActionToken privado() {
      Necessário para deserializar da JWT
      super();
   }
}
```

Se o token de ação que você está implementando contiver quaisquer campos personalizados que devem ser serializáveis para campos JSON, você deve considerar a implementação de um descendente de org.keycloak.representations.JsonWebToken classe que implementaria org.keycloak.models.actionTokenKeyModel interface. Nesse caso, você pode aproveitar a classe org.keycloak.authentication.actiontoken.defaultActionToken, pois já satisfaz ambas as condições, e usá-la diretamente ou implementar seu filho, os campos dos quais podem ser anotados com anotações apropriadas de Jackson, por exemplo, com.fasterxml.jackson.annotation.JsonProperty para serializá-las para JSON.

O exemplo a seguir estende o DemoActionToken do exemplo anterior com o demo-id decampo:

```
import com.fasterxml.jackson.anotação.JsonProperty;
import org.keycloak.authentication.actiontoken.DefaultActionToken;

demoActionToken de classe pública estende DefaultActionToken {

TOKEN_TYPE final de string estática pública = "my-demo-token";

JSON_FIELD_DEMO_ID de corda final estática privada = "demo-id";
```

```
   @JsonProperty(valor = JSON_FIELD_DEMO_ID)
demoId de string privado;

público DemoActionToken (Usuário de stringId,  int  absoluteExpirationInSecs, String
compoundAuthenticationSessionId, String demoId) {
super(userId, TOKEN_TYPE, absoluteExpirationInSecs,
nulo,compostoAuthenticationSessionId);
      este.demoId = demoId;
   }

demoActionToken privado() {
      você deve ter este construtor privado para desseializador
   }

   cadeia pública getDemoId() {
demoId de retorno;
   }
}
```

Aulas e implantação de embalagens

Para conectar seu próprio token de ação e seu manipulador, você precisa implementar poucas interfaces no lado do servidor:

- org.keycloak.authentication.actiontoken.ActionTokenHandler - manipulador real do token de ação para uma determinada ação (ou seja, por um determinado valor do campo de token de digitação).

  O método central nessa interface é o handleToken (token, contexto) que define a operação real executada ao receber o token de ação. Geralmente é alguma alteração de notas de sessão de autenticação, mas geralmente pode ser arbitrária. Este método só é chamado se todos os verificadores (incluindo aqueles definidos no (contexto) getVerifierstiverem sucesso, e é garantido que o token seria da classe retornada pelo método getTokenClass().

  Para ser capaz de determinar se o token de ação foi emitido para a sessão de autenticação atual, conforme descrito no item 2 acima,o método para extrair o ID da sessão de autenticação deve ser declarado no método getAuthenticationSessionIdFromToken (token, contexto). A implementação no DefaultActionToken retorna o valor do campo asid do token se ele for definido. Observe que você pode substituir esse método para retornar o ID atual da sessão de autenticação, independentemente do token - dessa forma, você pode criar tokens que entrariam no fluxo de autenticação em andamento antes que qualquer fluxo de autenticação fosse iniciado.

  Se a sessão de autenticação do token não corresponder à atual, o manipulador de tokens de ação será solicitado a iniciar um novo, chamando startFreshAuthenticationSession (token, contexto). Ele pode lançar um

VerificationException (ou melhor sua variante mais descritiva ExplainedTokenVerificationException) para sinalizar que seria proibido.

O manipulador de tokens também determina via método canUseTokenRepeatedly (token, contexto) se o token seria invalidado após o uso e a autenticação é concluída. Observe que se você tivesse um fluxo utilizando vários tokens de ação, apenas o último token seria invalidado. Nesse caso, você deve usar org.keycloak.models.ActionTokenStoreProvider no manipulador de tokens de ação para invalidar os tokens usados manualmente.

A implementação padrão da maioria dos métodos ActionTokenHandler é a classe abstrata org.keycloak.authentication.actiontoken.AbstractActionTokenHander no módulo de serviços keycloak. O único método que precisa ser implementado é o handleToken (token, contexto) que executa a ação real.

- org.keycloak.authentication.actiontoken.ActionTokenHandlerFactory - fábrica que instancia o manipulador de tokens de ação. As implementações devem substituir o valor getId() para retornar que deve corresponder precisamente ao valor do campo de digitação no token de ação.

  Observe que você tem que registrar a implementação personalizada do ActionTokenHandlerFactory, conforme explicado na seção Interfaces do Provedor de Serviços deste guia.

SPI do ouvinte de eventos

A redação de um Provedor de Ouvintes de Eventos começa implementando as interfaces EventListenerProvider e EventListenerProviderFactory. Consulte o Javadoc e exemplos para obter detalhes completos sobre como fazer isso.

Para obter detalhes sobre como empacotar e implantar um provedor personalizado, consulte o capítulo Interfaces do Provedor de Serviços.

SPI de mapeamento de papéis saml

Keycloak define um SPI para mapear funções saml em funções que existem no ambiente de SP. Os papéis retornados por um IDP de terceiros podem nem sempre corresponder aos papéis que foram definidos para a aplicação de SP, por isso há a necessidade de um mecanismo que permita mapear os papéis do SAML em diferentes funções. É usado pelo adaptador SAML depois de extrair os papéis da afirmação SAML para configurar o contexto de segurança do contêiner.

O SPI org.keycloak.adapters.saml.RoleMappingsProvider não impõe restrições aos mapeamentos que podem ser realizados. As implementações podem não apenas mapear funções em outras funções, mas também adicionar ou remover funções (e, assim, aumentar ou reduzir o conjunto de funções atribuídas ao principal SAML) dependendo do caso de uso.

Para obter detalhes sobre a configuração do provedor de mapeamento de funções para o adaptador SAML, bem como uma descrição das implementações padrão disponíveis, consulte o Guia de Aplicativos e Serviços de Garantia.

Implementação de um provedor de mapeamento de papéis personalizado

Para implementar um provedor de mapeamento de papéis personalizados, primeiro é necessário implementar a interface org.keycloak.adapters.saml.RoleMappingsProvider. Em seguida, um arquivo META-INF/services/org.keycloak.adapters.saml.RoleMappingsProvider contendo o nome totalmente qualificado da implementação personalizada deve ser adicionado ao arquivo que também contém a classe de implementação. Este arquivo pode ser:

- O arquivo WAR do aplicativo SP onde a classe do provedor está incluída em web-INF/classes;
- Um arquivo JAR personalizado que será adicionado ao WEB-INF/lib da GUERRA de aplicativos de SP;
- (Somente WildFly/JBoss EAP) Um arquivo JAR personalizado configurado como um módulo jboss e referenciado na estrutura de implantação jboss.xml do aplicativo DE SP WAR.

Quando o aplicativo sp é implantado, o provedor de mapeamento de papéis que será usado é selecionado pelo id que foi definido em keycloak-saml.xml ou no subsistema keycloak-saml. Assim, para habilitar seu provedor personalizado basta certificar-se de que seu id está devidamente definido na configuração do adaptador.

SPI de armazenamento do usuário

Você pode usar o SPI de armazenamento do usuário para gravar extensões ao Keycloak para se conectar a bancos de dados de usuários externos e lojas de credenciais. O suporte integrado de LDAP e ActiveDirectory é uma implementação deste SPI em ação. Fora da caixa, o Keycloak usa seu banco de dados local para criar, atualizar e procurar usuários e validar credenciais. Muitas vezes, porém, as organizações têm bancos de dados de usuários proprietários externos existentes que não podem migrar para o modelo de dados do Keycloak. Para essas situações, os desenvolvedores de aplicativos podem escrever implementações do SPI de armazenamento do usuário para fazer a

ponte entre a loja de usuários externo e o modelo interno de objeto do usuário que o Keycloak usa para fazer login nos usuários e gerenciá-los.

Quando o tempo de execução do Keycloak precisa procurar um usuário, como quando um usuário está fazendo login, ele executa uma série de etapas para localizar o usuário. Primeiro parece ver se o usuário está no cache do usuário; se o usuário for encontrado, ele usa essa representação na memória. Em seguida, ele procura o usuário dentro do banco de dados local Keycloak. Se o usuário não for encontrado, ele então faz loops através das implementações do provedor de SPI de armazenamento do usuário para realizar a consulta do usuário até que um deles retorne o usuário que o tempo de execução está procurando. O provedor consulta a loja de usuário externo para o usuário e mapeia a representação de dados externos do usuário ao metamodel do usuário da Keycloak.

As implementações do provedor de SPI de armazenamento do usuário também podem executar consultas de critérios complexos, executar operações CRUD em usuários, validar e gerenciar credenciais ou executar atualizações em massa de muitos usuários ao mesmo tempo. Depende das capacidades da loja externa.

As implementações do provedor de SPI de armazenamento do usuário são embaladas e implantadas de forma semelhante aos (e muitas vezes são) componentes Java EE. Eles não estão habilitados por padrão, mas devem ser ativados e configurados por reino sob a guia `Federação` do Usuário no console de administração.

> Se a implementação do provedor de usuário estiver usando alguns atributos do usuário como atributos de metadados para vincular/estabelecer a identidade do usuário, certifique-se de que os usuários não são capazes de editar os atributos e os atributos correspondentes são somente lidos. O exemplo é o atributo `LDAP_ID`, que o provedor LDAP Keycloak embutido está usando para armazenar o ID do usuário no lado do servidor LDAP. Veja os detalhes no capítulo de mitigação do modelo Ameaça.

Interfaces de provedor

Ao construir uma implementação do SPI de armazenamento do usuário, você tem que definir uma classe de provedor e uma fábrica de provedores. As instâncias da classe do provedor são criadas por transação por fábricas de provedores. As classes de provedores fazem todo o levantamento pesado da procuração do usuário e outras operações de usuário. Eles devem implementar a interface `org.keycloak.storage.UserStorageProvider`.

```
pacote org.keycloak.storage;

interface pública UserStorageProvider estende Provedor {

  /**
```

```
    * Ligue de volta quando um reino for removido.  Implemente isso se, por exemplo, você
quiser fazer algum
    * limpeza no armazenamento do usuário quando um reino é removido
    *
    @param reino
    */
   inadimplência
   vazio preRemove (Reinomodel) {

   }

   /**
    * Ligue de volta quando um grupo for removido.  Permite que você faça coisas como
remover um usuário
    * mapeamento de grupo em sua loja externa, se for o caso
    *
    @param reino
    * @param grupo
    */
   inadimplência
   vazio preRemove (Reinommodel, grupo GroupModel) {

   }

   /**
    * Ligue de volta quando um papel for removido.  Permite que você faça coisas como
remover um usuário
    * Mapeamento de papéis em sua loja externa, se for o caso

    @param reino
    * @param papel
    */
   inadimplência
   vazio preRemove (ReinoModel, Papel Modelo) {

   }

}
```

Você pode estar pensando que a interface `UserStorageProvider` é bastante escassa? Você verá mais tarde neste capítulo que existem outras interfaces de mix-in que sua classe de provedor pode implementar para suportar a carne da integração do usuário.

As instâncias do `UserStorageProvider` são criadas uma vez por transação. Quando a transação estiver concluída, o método `UserStorageProvider.close()` é invocado e a instância é então coletada com lixo. As instâncias são criadas por fábricas de provedores. As fábricas de provedores implementam a interface `org.keycloak.storage.UserStorageProviderFactory`.

```
pacote org.keycloak.storage;
```

```
/**
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
interface pública UsuárioStorageProviderFactory<T estende UserStorageProvider> estende
ComponentFactory<T, UserStorageProvider> {

   /**
    * Este é o nome do provedor e será mostrado no console administrativo como uma opção.
    *
    @return
    */
   @Override
   GetId de cordas();

   /**
    * chamado por transação Keycloak.
    *
    * sessão de @param
    * @param modelo
    @return
    */
   T criar (Sessão KeycloakSession, Modelo Desmodel de Componentes);
...
}
```

As classes de fábrica do provedor devem especificar a classe de provedor de concreto como um parâmetro de modelo ao implementar o UserStorageProviderFactory. Isso é uma obrigação, pois o tempo de execução introspectará esta classe para verificar suas capacidades (as outras interfaces que implementa). Então, por exemplo, se a sua classe de provedor for chamada FileProvider, então a classe de fábrica deve ser assim:

```
Classe pública FileProviderFactory implementa UserStorageProviderFactory<FileProvider> {

público String getId() { devolução "file-provider"; }

criar fileProvider público (Sessão KeycloakSession, ComponentModel modelo) {
     ...
   }
```

O método getId() retorna o nome do provedor de armazenamento do usuário. Este id será exibido na página da Federação de Usuários do console administrativo quando você quiser habilitar o provedor para um reino específico.

O método create() é responsável por alocar uma instância da classe provedora. É preciso um parâmetro org.keycloak.models.KeycloakSession. Este objeto pode ser usado para procurar outras informações e metadados, bem como fornecer acesso a vários outros componentes dentro do tempo de execução. O parâmetro ComponentModel representa como o provedor foi ativado e configurado dentro de um reino específico. Ele contém o

id de instância do provedor habilitado, bem como qualquer configuração que você possa ter especificado para ele quando você habilitado através do console administrativo.

O `UserStorageProviderFactory` também tem outros recursos que passaremos mais tarde neste capítulo.

Interfaces de capacidade do provedor

Se você examinou a interface `UserStorageProvider` de perto, você pode notar que ela não define nenhum método para localizar ou gerenciar usuários. Esses métodos são realmente definidos em outras *interfaces* de capacidade, dependendo do escopo de recursos que sua loja de usuário externa pode fornecer e executar. Por exemplo, algumas lojas externas são somente leitura e só podem fazer consultas simples e validação de credenciais. Você só será obrigado a implementar as *interfaces de capacidade* para os recursos que você é capaz de fazer. Você pode implementar essas interfaces:

Spi	**descrição**
org.keycloak.storage.user.UserLookupProvider	Esta interface é necessária se você quiser ser capaz de fazer login com os usuários desta loja externa. A maioria dos provedores (todos?) implementa essa interface.
org.keycloak.storage.user.UserQueryProvider	Define consultas complexas que são usadas para localizar um ou mais usuários. Você deve implementar esta interface se quiser visualizar e gerenciar usuários a partir do console de administração.
org.keycloak.storage.user.UserRegistrationProvider	Implemente esta interface se o provedor suportar a adição e a remoção de usuários.
org.keycloak.storage.user.UserBulkUpdateProvider	Implemente esta interface se o provedor suportar a atualização em massa de um conjunto de usuários.
org.keycloak.credential.CredentialInputValidator	Implemente essa interface se o provedor puder validar um ou mais tipos de credenciais diferentes (por exemplo, se o provedor puder validar uma senha).

Spi	descrição
org.keycloak.credential.CredentialInputUpdater	Implemente esta interface se o provedor suportar a atualização de um ou mais tipos de credenciais diferentes.

Interfaces de modelo

A maioria dos métodos definidos nas *interfaces* de capacidade retornam ou são passados em representações de um usuário. Essas representações são definidas pela interface org.keycloak.models.UserModel. Os desenvolvedores de aplicativos são obrigados a implementar essa interface. Ele fornece um mapeamento entre a loja de usuário externo e o metamodel do usuário que o Keycloak usa.

```
pacote org.keycloak.models;

interface pública UserModel estende RoleMapperModel {
   GetId de cordas();

   String getUsername();
conjunto vazioSa nome de usuário (nome de usuário de string);

   String getFirstName();
conjunto de vazioFirstName(String firstName);

   String getLastName();
conjunto vazioLastname (String lastName);

   String getEmail();
void setEmail (String email);
...
}
```

As implementações do UserModel fornecem acesso a metadados de leitura e atualização sobre o usuário, incluindo coisas como nome de usuário, nome, e-mail, funções e mapeamentos de grupos, bem como outros atributos arbitrários.

Existem outras classes de modelos dentro do pacote org.keycloak.models que representam outras partes do metamodel Keycloak: RealmModel, RoleModel, GroupModele ClientModel.

IDs de armazenamento

Um método importante do UserModel é o método getId(). Ao implementar os desenvolvedores do UserModel devem estar cientes do formato de identificação do usuário. O formato deve ser:

```
"f:" + id componente + ":" + id externo
```

O tempo de execução do Keycloak muitas vezes tem que procurar usuários por sua identificação de usuário. O id do usuário contém informações suficientes para que o tempo de execução não precise consultar cada `UsuárioStorageProvider` no sistema para encontrar o usuário.

O id componente é o id retornado do `ComponentModel.getId()`. O `ComponentModel` é passado como um parâmetro ao criar a classe provedora para que você possa obtê-lo a partir daí. O id externo é uma informação que sua classe de provedor precisa encontrar o usuário na loja externa. Este é muitas vezes um nome de usuário ou um uid. Por exemplo, pode ser algo assim:

```
f:332a234e31234:wburke
```

Quando o tempo de execução faz uma busca por id, o id é analisado para obter o id componente. O id de componente é usado para localizar o `UserStorageProvider` que foi originalmente usado para carregar o usuário. Esse provedor é então passado a id. O provedor analisa novamente o id para obter o id externo e ele usará para localizar o usuário no armazenamento externo do usuário.

Embalagem e implantação

Os provedores de armazenamento de usuário são embalados em um JAR e implantados ou não para o tempo de execução do Keycloak da mesma forma que você implantaria algo no servidor de aplicativos WildFly. Você pode copiar o JAR diretamente para o diretório `autônomo/implantações/diretório` do servidor ou usar o CLI JBoss para executar a implantação.

Para que o Keycloak reconheça o provedor, você precisa adicionar um arquivo ao JAR: `META-INF/services/org.keycloak.storage.UserStorageProviderFactory`. Este arquivo deve conter uma lista separada de linha de nomes de classe totalmente qualificados das implementações `UserStorageProviderFactory`:

```
org.keycloak.examples.federation.properties.ClasspathPropertiesStorageFactory
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

Keycloak suporta a implantação quente desses JARs provedores. Você também verá mais tarde neste capítulo que você pode empacotá-lo dentro e como componentes Java EE.

Simples leitura apenas, exemplo de procura

Para ilustrar o básico da implementação do SPI de armazenamento do usuário, vamos passar por um exemplo simples. Neste capítulo você verá a implementação de um simples UserStorageProvider que procura os usuários em um simples arquivo de propriedade. O arquivo de propriedade contém definições de nome de usuário e senha e é codificado para um local específico no classpath. O provedor poderá procurar o usuário por ID e nome de usuário e também ser capaz de validar senhas. Os usuários originários deste provedor serão somente lidos.

Classe provedora

A primeira coisa pela que vamos passar é a classe UserStorageProvider.

```
classe pública PropertyFileUserStorageProvider implementa
      UserStorageProvider,
      UserLookupProvider,
      CredencialInputValidator,
      CredencialInputUpdater
{
...
}
```

Nossa classe de provedor, PropertyFileUserStorageProvider,implementa muitas interfaces. Ele implementa o UserStorageProvider, pois isso é um requisito base do SPI. Ele implementa a interface UserLookupProvider porque queremos poder fazer login com usuários armazenados por este provedor. Ele implementa a interface CredentialInputValidator porque queremos ser capazes de validar senhas inseridas usando a tela de login. Nosso arquivo de propriedade é somente leitura. Implementamos o CredentialInputUpdater porque queremos postar uma condição de erro quando o usuário tenta atualizar sua senha.

```
sessão de sessão de sessão de keycloaksession protegida;
propriedades protegidas;
modelo de modelo de modelo de componente protegido;
  mapa de usuários carregados nesta transação
Mapa protegido<String, UserModel> carregadosUsers = novo HashMap<>();

propriedade públicaFileUserStorageProvider(sessão KeycloakSession, ComponentModel
model, propriedades propriedades) {
      estasessão .session = sessão;
      estemodelo .= modelo;
      estas.propriedades = propriedades;
  }
```

O construtor desta classe provedor vai armazenar a referência ao KeycloakSession, ComponentModele arquivo de propriedade. Usaremos tudo isso mais tarde. Observe também que há um mapa de usuários carregados. Sempre que encontrarmos um usuário, vamos armazená-lo neste mapa para evitarmos recriá-lo novamente dentro da mesma transação. Essa é uma boa prática a seguir, pois muitos provedores precisarão fazer isso (ou seja, qualquer provedor que se integre ao JPA). Lembre-se também que as

instâncias da classe do provedor são criadas uma vez por transação e são encerradas após o fim da transação.

Implementação do UserLookupProvider

```
   @Override
Público UserModel getUserByUsername (String username, RealmModel realm) {
      Adaptador UserModel = loadedUsers.get (nome de usuário);
se (adaptador == nulo){
         Senha de string = propriedades.getProperty (nome de usuário);
se (senha != nulo){
            adaptador = criarAdapter(reino, nome de usuário);
            loadedUsers.put (nome de usuário, adaptador);
         }
      }
adaptador de retorno;
   }

UserModel protegido criarAdapter (RealmModel realm, String username) {
      retornar novo AbstractUserAdapter(sessão, reino, modelo) {
         @Override
         cadeia pública getUsername() {
            nome de usuário de retorno;
         }
      };
   }

   @Override
Público UserModel getUserById(String id, RealmModel realm) {
Armazenamento De armazenamentoId = novo StorageId(id);
      Nome de usuário de string = storageId.getExternalId();
      retorno getUserByUsername (nome de usuário, reino);
   }

   @Override
Público UserModel getUserByEmail (String email, RealmModel realm) {
retorno nulo;
   }
```

O método `getUserByUsername()` é invocado pela página de login keycloak quando um usuário faz login. Em nossa implementação, primeiro verificamos o mapa `carregadosusUs` para ver se o usuário já foi carregado dentro desta transação. Se não foi carregado, procuramos no arquivo da propriedade o nome de usuário. Se ele existir, criamos uma implementação do `UserModel`, armazene-o em `carregamentosuus` para referência futura e retornemos esta instância.

O método `createAdapter()` usa a classe helper `org.keycloak.storage.adapter.AbstractUserAdapter`. Isso fornece uma implementação base para o `UserModel`. Ele gera automaticamente um id de usuário com base no formato de identificação de armazenamento necessário usando o nome de usuário do usuário como o id externo.

```
"f:" + id de componente + + + nome de usuário
```

Cada método get de `AbstractUserAdapter` retorna coleções nulas ou vazias. No entanto, os métodos que retornam os mapeamentos de funções e grupos retornarão as funções e grupos padrão configurados para o reino para cada usuário. Cada método definido de `AbstractUserAdapter` lançará um `org.keycloak.storage.ReadOnlyException`. Então, se você tentar modificar o usuário no console de administração, você terá um erro.

O método `getUserById()` analisa o parâmetro de `identificação` usando a classe de ajudante `org.keycloak.storage.StorageId`. O método `StorageId.getExternalId()` é invocado para obter o nome de usuário incorporado no parâmetro `id`. O método então delega para `obterUserByUsername()`.

Os e-mails não são armazenados, portanto, o método `getUserByEmail()` retorna nulo.

Implementação do CredencialInputValidator

Em seguida, vamos olhar para as implementações do método para `CredentialInputValidator`.

```
   @Override
   público boolean isConfiguredFor (RealmModel realm, UserModel user, String
credentialType) {
      Senha de string = propriedades.getProperty (user.getUsername());
devolução credencialType.equals(CredentialModel.PASSWORD) && senha != nulo;
   }

   @Override
suportes booleanos públicosCredentialType (String credentialType) {
      credencial de retornoType.equals (CredentialModel.PASSWORD);
   }

   @Override
   público boolean isValid (RealmModel realm, UserModel user, CredentialInput input) {
se (!suportaCredentialType(input.getType()) retornar falso;

      Senha de string = propriedades.getProperty (user.getUsername());
se (senha == nulo) retornar falso;
      return password.equals (input.getChallengeResponse());
   }
```

O método `isConfiguredFor()` é chamado pelo tempo de execução para determinar se um tipo de credencial específico está configurado para o usuário. Este método verifica se a senha está definida para o usuário.

O método `supportsCredentialType()` retorna se a validação é suportada para um tipo de credencial específica. Verificamos se o tipo de credencial é `senha`.

O método `isValid()` é responsável pela validação de senhas. O parâmetro `CredentialInput` é realmente apenas uma interface abstrata para todos os tipos de credenciais. Nós nos

certificamos de que suportamos o tipo de credencial e também que ele é uma instância do `UserCredentialModel`. Quando um usuário faz login através da página de login, o texto simples da entrada de senha é colocado em uma instância do `UserCredentialModel`. O método `isValid()` verifica esse valor em relação à senha de texto simples armazenada no arquivo de propriedades. Um valor de devolução de `verdade` significa que a senha é válida.

Implementação do CredencialInputUpdater

Como observado anteriormente, a única razão pela qual implementamos a interface `CredentialInputUpdater` neste exemplo é para proibir modificações de senhas de usuário. A razão pela qual temos que fazer isso é porque, caso contrário, o tempo de execução permitiria que a senha fosse substituída no armazenamento local keycloak. Falaremos mais sobre isso mais tarde neste capítulo.

```
   @Override
   Atualização booleana públicaCredential (RealmModel realm, UserModel user,
CredentialInput input) {
se (input.getType().equals(CredentialModel.PASSWORD)) lançar novo ReadOnlyException("o
usuário é lido apenas para esta atualização");

retorno falso;
   }

   @Override
   vazio público desativarCredentialType (ReinomModel, usuário do UserModel,
Credenciamento de String) {

   }

   @Override
conjunto público<String> getDisableCredentialTypes (RealmModel realm, UserModel user) {
      Collections.EMPTY_SET de retorno;
   }
```

O método `updateCredential()` apenas verifica se o tipo de credencial é senha. Se for, um `ReadOnlyException` é lançado.

Implementação da fábrica do provedor

Agora que a classe de provedores está completa, agora voltamos nossa atenção para a classe de fábrica de provedores.

```
classe pública PropertyFileUserStorageProviderFactory
            implementa UserStorageProviderFactory<PropertyFileUserStorageProvider> {

PROVIDER_NAME final de string estática pública = "readonly property-file";

   @Override
público String getId() {
```

```
    PROVIDER_NAME de retorno;
  }
```

A primeira coisa a notar é que, ao implementar a classe `UserStorageProviderFactory`, você deve passar na implementação da classe de provedor de concreto como um parâmetro de modelo. Aqui especificamos a classe de provedor que definimos antes: `PropertyFileUserStorageProvider`.

> Se você não especificar o parâmetro de modelo, seu provedor não funcionará. O tempo de execução faz introspecção de classe para determinar as *interfaces de capacidade* que o provedor implementa.

O método `getId()` identifica a fábrica no tempo de execução e também será a sequência mostrada no console administrativo quando você quiser habilitar um provedor de armazenamento de usuário para o reino.

inicialização

```
  logger final estático privado = Logger.getLogger
(PropertyFileUserStorageProviderFactory.class);
propriedades protegidas = novas propriedades();

  @Override
  vazio público init (Config.Scope config) {
    InputStream é = getClass().getClassLoader().getResourceAsStream("/users.properties");

se (é == nulo){
        logger.warn("Não foi possível encontrar usuários.propriedades em classpath");
} outra coisa {
        tente {
          propriedades.carga(é);
} catch (IOException ex) {
logger.error("Falhaao carregar o arquivo user.properties",ex);
        }
    }
  }

  @Override
propriedade públicaFileUserStorageProvider create (KeycloakSession session,
ComponentModel model) {
    devolver novo PropertyFileUserStorageProvider (sessão, modelo, propriedades);
  }
```

A interface `UserStorageProviderFactory` tem um método `init()` opcional que você pode implementar. Quando o Keycloak inicializado, apenas uma instância de cada fábrica de provedores é criada. Também na hora do inicial, o método `init()` é chamado em cada uma dessas instâncias de fábrica. Há também um método `postInit()` que você pode implementar também. Depois que o método `init()` de cada fábrica é invocado, seus métodos `postInit()` são chamados.

Em nossa implementação do método `init()`, encontramos o arquivo de propriedade contendo nossas declarações de usuário da classpath. Em seguida, carregamos o campo `de propriedades` com as combinações de nome de usuário e senha armazenadas lá.

O parâmetro `Config.Scope` é uma configuração de fábrica que pode ser configurada em `.xml`autônomos, `autônomo-ha.xml`ou `domínio.xml`.

Por exemplo, adicionando o seguinte a `autônomo.xml`:

```
nome <spi ="armazenamento">
<provo nome="readonly property-file"  ativado="verdadeiro">
      <propriedades>
< nomeda propriedade= valor"caminho" ="/outros usuários.propriedades"/>
      </propriedades>
   </provedor>
</spi>
```

Podemos especificar o classpath do arquivo de propriedade do usuário em vez de codificar ele. Em seguida, você pode recuperar a configuração no `PropertyFileUserStorageProviderFactory.init()`:

```
vazio público init (Config.Scope config) {
   Caminho de cordas = config.get("caminho");
   InputStream é = getClass().getClassLoader().getResourceAsStream(path);

   ...
}
```

Criar método

Nosso último passo na criação da fábrica de provedores é o método `create()`.

```
   @Override
propriedade públicaFileUserStorageProvider create (KeycloakSession session,
ComponentModel model) {
      devolver novo PropertyFileUserStorageProvider (sessão, modelo, propriedades);
   }
```

Nós simplesmente alocamos a classe `PropertyFileUserStorageProvider`. Este método de criação será chamado uma vez por transação.

Embalagem e implantação

Os arquivos de classe para a implementação do nosso provedor devem ser colocados em um frasco. Você também tem que declarar a classe de fábrica do provedor dentro do arquivo `META-INF/services/org.keycloak.storage.UserStorageProviderFactory`.

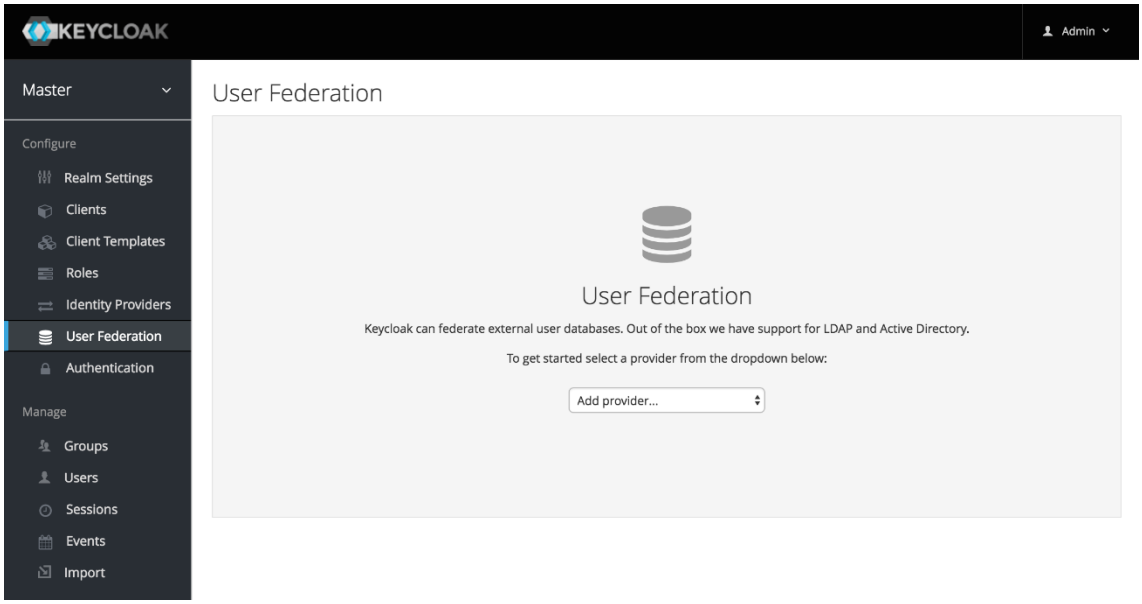
```
org.keycloak.examples.federation.properties.FilePropertiesStorageFactory
```

Depois de criar o frasco, você pode implantá-lo usando meios regulares do WildFly: copie o frasco no diretório autônomo/implantações/ou usando o CLI JBoss.

Habilitação do Provedor no Console de Administração

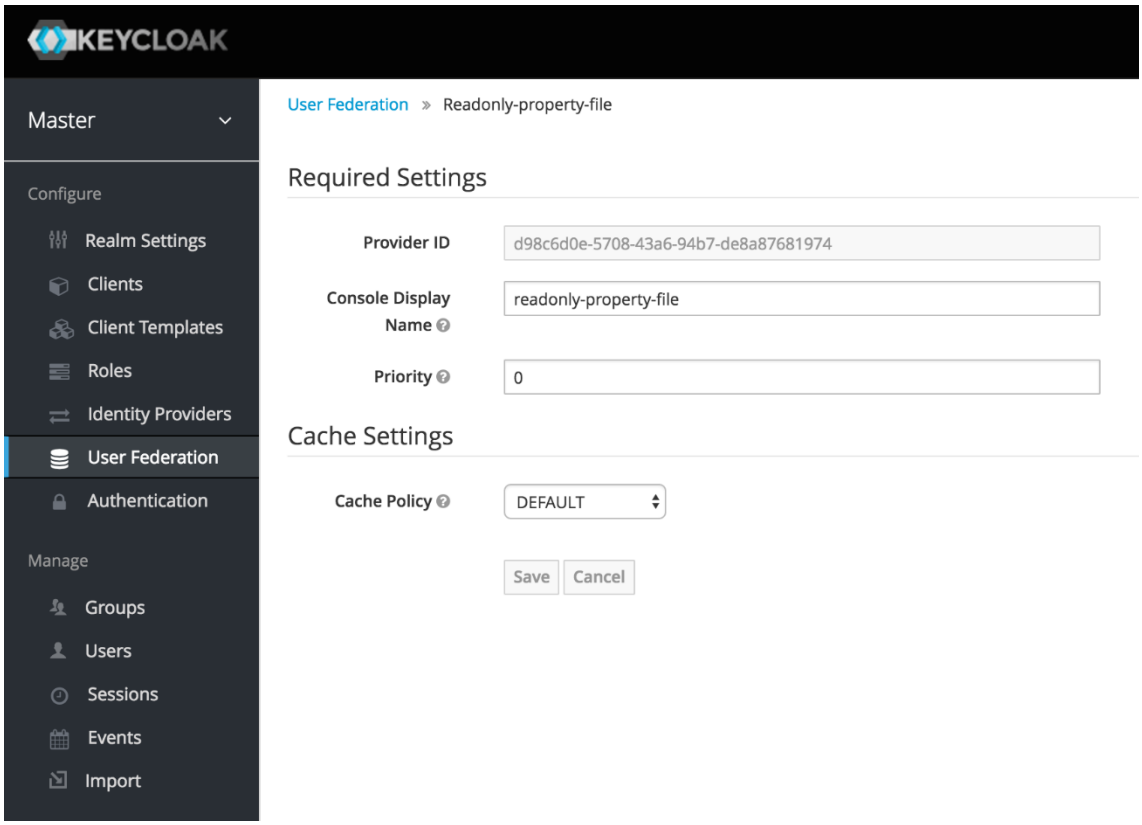
Você habilita os provedores de armazenamento de usuários por reino dentro da página da Federação do Usuário no console de administração.

Federação dos Usuários



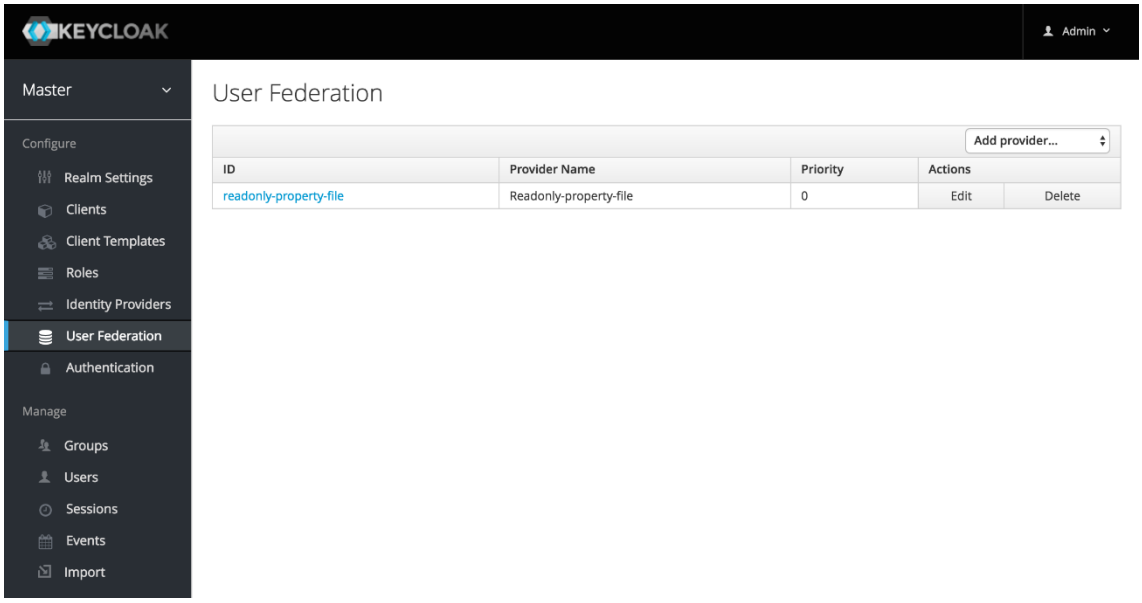
Selecione o provedor que acabamos de criar na lista: readonly property-file. Ele traz você para a página de configuração para o nosso provedor. Não temos nada para configurar, então clique em **Salvar**.

Provedor configurado



Quando você voltar para a página principal da Federação de Usuários, agora você vê o seu provedor listado.

Federação dos Usuários



Agora você poderá fazer login com um usuário declarado no arquivo users.properties. Este usuário só poderá visualizar a página da conta após o login.

Técnicas de configuração

Nosso exemplo de PropertyFileUserStorageProvider é um pouco inventado. Ele é codificado com força para um arquivo de propriedade que está incorporado no frasco do provedor, o que não é terrivelmente útil. Podemos querer tornar a localização deste arquivo configurável por instância do provedor. Em outras palavras, podemos querer reutilizar este provedor várias vezes em vários reinos diferentes e apontar para arquivos de propriedade de usuários completamente diferentes. Também queremos executar esta configuração dentro da interface do console de administração.

O UserStorageProviderFactory tem métodos adicionais que você pode implementar que lidam com a configuração do provedor. Você descreve as variáveis que deseja configurar por provedor e o console de administração renderiza automaticamente uma página de entrada genérica para coletar essa configuração. Quando implementados, os métodos de retorno de chamada também validam a configuração antes de ser salva, quando um provedor é criado pela primeira vez e quando é atualizado. UserStorageProviderFactory herda esses métodos da interface org.keycloak.component.ComponentFactory.

```
   Lista<ProviderConfigProperty> obterConfigProperties();

   inadimplência
   vazio validaConfiguração (Sessão keycloakSession, reino RealmModel, ComponentModel
modelo)
         lança ComponentValidationException
   {

   }

   inadimplência
   vazio noCreate (Sessão KeycloakSession, Reino DomModel, Modelo Desom) {

   }

   inadimplência
   vazio onUpdate (Sessão KeycloakSession, ReinomModel, Modelo Desom) {

   }
```

O método ComponentFactory.getConfigProperties() retorna uma lista de instâncias org.keycloak.provider.ProviderConfigProperty. Essas instâncias declaram metadados necessários para renderizar e armazenar cada variável de configuração do provedor.

Exemplo de configuração

Vamos expandir nosso exemplo De PropriedadeFileUserStorageProviderFactory para permitir que você aponte uma instância de provedor para um arquivo específico em disco.

PropriedadeFileUserStorageProviderFactory

```
classe pública PropertyFileUserStorageProviderFactory
            implementa UserStorageProviderFactory<PropertyFileUserStorageProvider> {

   Lista final estática protegida<ProviderConfigProperty> configMetadata;

   estática {
      configMetadata = ProviderConfigurationBuilder.create()
            .property().nome("caminho")
            .tipo (ProviderConfigProperty.STRING_TYPE)
.label("Path")
.defaultValue(${jboss.server.config.dir}/example-users.properties")
.helpText("Caminho do arquivo dearquivo para propriedades")
            .add().build();
   }

   @Override
Lista pública<ProviderConfigProperty> getConfigProperties() {
      retorno configMetadata;
   }
```

A classe ProviderConfigurationBuilder é uma ótima classe de ajudantes para criar uma lista de propriedades de configuração. Aqui especificamos uma variável nomeada caminho que é um tipo string. Na página de configuração do console de administração para este provedor, essa variável de configuração é rotulada como Path e tem um valor padrão de ${jboss.server.config.dir}/example-users.properties. Quando você passa o mouse sobre a ponta de ferramenta desta opção de configuração, ele exibe o texto de ajuda, o caminho do arquivo do arquivo de propriedades.

A próxima coisa que queremos fazer é verificar se esse arquivo existe em disco. Não queremos habilitar uma instância deste provedor no reino, a menos que ele aponte para um arquivo de propriedade de usuário válido. Para isso, implementamos o método validEguração().

```
   @Override
   vazio público validaConfiguração (Sessão keycloakSession, reino RealmModel,
ComponentModel config)
            lança ComponentValidationException {
String fp = config.getConfig().getFirst("path");
se (fp ==  nulo) lançar novo  ComponentValidationException ("arquivode propriedade do
usuário nãoexiste");
      fp = EnvUtil.replace(fp);
Arquivo de arquivo = novo  Arquivo(fp);
      se (!file.existe()) {
         lançar novo ComponentValidationException("arquivo de propriedade do usuário não
existe");
      }
   }
```

No método validEguração() o obteremos a variável de configuração do ComponentModel e verificamos se esse arquivo existe em disco. Observe que usamos o

método org.keycloak.common.util.EnvUtil.replace(). Com este método, qualquer string que tenha ${} dentro dele substituirá isso por um valor de propriedade do sistema. A string ${jboss.server.config.dir} corresponde à configuração/diretório do nosso servidor e é realmente útil para este exemplo.

A próxima coisa que temos que fazer é remover o método antigo init(). Fazemos isso porque os arquivos de propriedade do usuário serão únicos por instância do provedor. Movemos essa lógica para o método de criação().

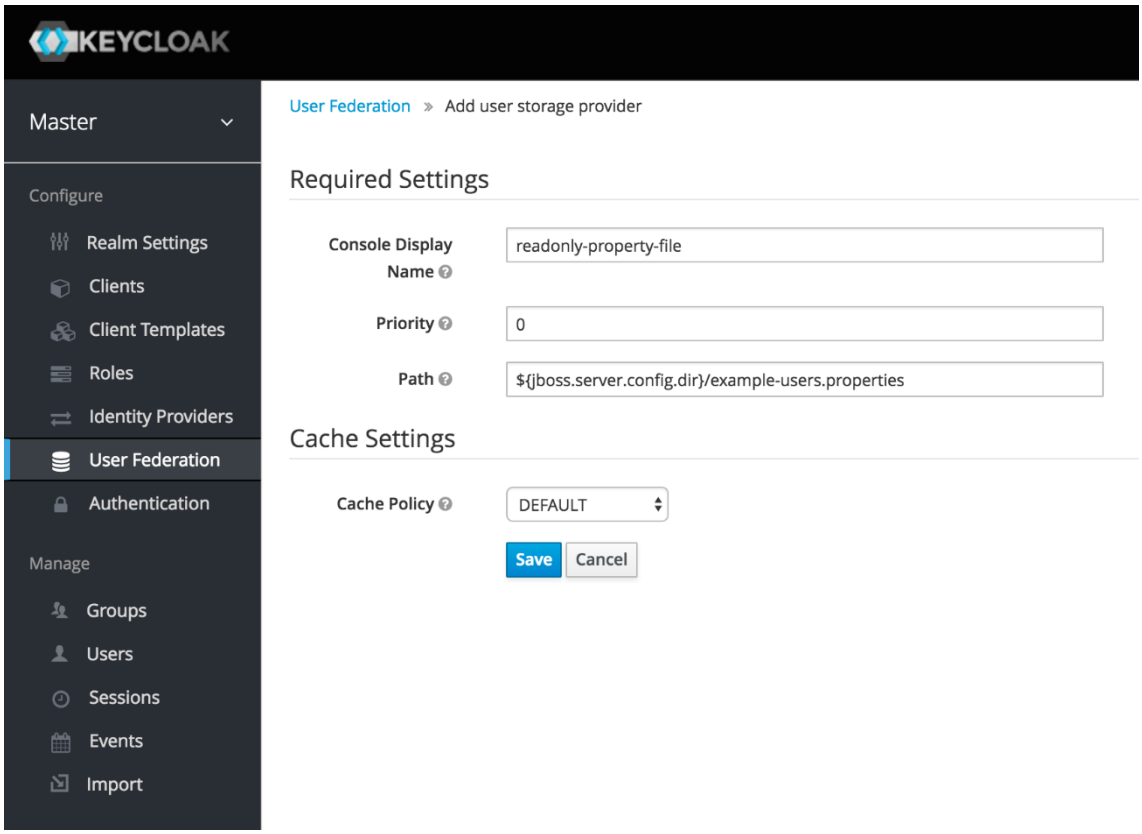
```
   @Override
propriedade públicaFileUserStorageProvider create (KeycloakSession session,
ComponentModel model) {
Caminho de string = model.getConfig().getFirst("path");

Adereços de propriedades = novas  propriedades();
      tente {
InputStream é = novo  FileInputStream (caminho);
         props.load(is);
         is.close();
} catch  (IOException e) {
         lançar novo RuntimeException(e);
      }

      devolver novo PropertyFileUserStorageProvider(sessão, modelo, adereços);
   }
```

Essa lógica é, naturalmente, ineficiente à medida que cada transação lê todo o arquivo de propriedade do usuário a partir do disco, mas espero que isso ilustre, de forma simples, como enganchar em variáveis de configuração.

Configuração do Provedor no Console de Administração

Agora que a configuração está ativada, você pode definir a variável de caminho ao configurar o provedor no console de administração.

Provedor configurado



Adicionar/remover interfaces de recursos de usuário e consulta

Uma coisa que não fizemos com nosso exemplo é permitir que ele adicione e remova usuários ou altere senhas. Os usuários definidos em nosso exemplo também não são consultados ou viáveis no console de administração. Para adicionar esses aprimoramentos, nosso provedor de exemplo deve implementar as interfaces `UserQueryProvider` e `UserRegistrationProvider`.

Implementação do UserRegistrationProvider

Para implementar a adição e remoção de usuários desta loja em particular, primeiro temos que ser capazes de salvar nosso arquivo de propriedades em disco.

PropertyFileUserStorageProvider

```
   vazio público salvar() {
Caminho de string = model.getConfig().getFirst("path");
      caminho = EnvUtil.replace(path);
      tente {
FileOutputStream fos = novo  FileOutputStream (caminho);
         properties.store(fos, "");
         fos.close();
} catch  (IOException e) {
```

```
            lançar novo RuntimeException(e);
        }
    }
```

Em seguida, a implementação dos métodos `addUser()` e `removeu()` torna-se simples.

PropertyFileUserStorageProvider

```
UNSET_PASSWORD final de sequênciaestática pública="#$!-UNSET-PASSWORD";

    @Override
Público UserModel addUser (RealmModel realm, String username) {
        sincronizados (propriedades) {
            propriedades.setProperty (nome de usuário, UNSET_PASSWORD);
            salvar();
        }
        retorno criarAdapter(reino, nome de usuário);
    }

    @Override
    público boolean removeUser (RealmModel realm, UserModel user) {
        sincronizados (propriedades) {
se (propriedades.remove (user.getUsername()) == nulo) devolvê-lo ;
            salvar();
retorno verdadeiro;
        }
    }
```

Observe que ao adicionar um usuário definimos o valor da senha do mapa da propriedade para ser `UNSET_PASSWORD`. Fazemos isso porque não podemos ter valores nulos para um imóvel no valor da propriedade. Também temos que modificar os métodos `credenciais para` refletir isso.

O método `addUser()` será chamado se o provedor implementar a interface `UserRegistrationProvider`. Se o provedor tiver um interruptor de configuração para desativar a adição de um usuário, o retorno `nulo` deste método pulará o provedor e chamará o próximo.

PropertyFileUserStorageProvider

```
    @Override
    público boolean isValid (RealmModel realm, UserModel user, CredentialInput input) {
se (!suportaCredentialType(input.getType()) || !( instância de entrada do
UserCredentialModel)) retorno falso;

        Credenciamento do UserCredentialModel = (UserCredentialModel)entrada;
        Senha de string = propriedades.getProperty (user.getUsername());
se (senha == nulo || UNSET_PASSWORD.equals (senha)) devolução falsa;
        return password.equals(cred.getValue());
    }
```

Uma vez que agora podemos salvar nosso arquivo de propriedade, também faz sentido permitir atualizações de senha.

PropertyFileUserStorageProvider

```
   @Override
   Atualização booleana públicaCredential (RealmModel realm, UserModel user,
CredentialInput input) {
se (!( instância de entrada  do  UserCredentialModel))  retorno  falso;
se (!input.getType().equals(CredentialModel.PASSWORD))  retornar  falso;
      Credenciamento do UserCredentialModel = (UserCredentialModel)entrada;
      sincronizados (propriedades) {
         properties.setProperty (user.getUsername(), cred.getValue());
         salvar();
      }
retorno verdadeiro;
   }
```

Agora também podemos implementar a desativação de uma senha.

PropertyFileUserStorageProvider

```
   @Override
   vazio público desativarCredentialType (ReinomModel, usuário do UserModel,
Credenciamento de String) {
se (!credencialType.equals(CredencialModel.PASSWORD))  retornar;
      sincronizados (propriedades) {
         propriedades.setProperty (user.getUsername(), UNSET_PASSWORD);
         salvar();
      }

   }

conjunto final estático privado<String> desabilitarTypes =  novo  HashSet<>();

   estática {
      desativarTypes.add (CredentialModel.PASSWORD);
   }

   @Override
conjunto público<String> getDisableCredentialTypes (RealmModel realm, UserModel user) {

desabilitar tipos;
   }
```

Com esses métodos implementados, agora você poderá alterar e desativar a senha para o usuário no console de administração.

Implementação do UserQueryProvider

Sem implementar o UserQueryProvider, o console de administração não seria capaz de visualizar e gerenciar usuários que foram carregados pelo nosso provedor de exemplo. Vamos ver a implementação desta interface.

PropertyFileUserStorageProvider

```
   @Override
   int público getUsersCount (RealmModel realm) {
propriedades de retorno.tamanho();
   }

   @Override
Lista pública<UserModel> getUsers (reino do RealmModel) {
retorno getUsers(reino, 0,Integer.MAX_VALUE);
   }

   @Override
Lista pública<UserModel> getUsers (RealmModel realm, int firstResult, int maxResults) {
Lista<UserModel> usuários = novo LinkedList<>();
int i = 0;
      para (Objeto obj : properties.keySet()) {
se (i++ < primeiroResult) continuar;
         Nome de usuário de string = (String)obj;
         UsuárioModel usuário = getUserByUsername (nome de usuário, reino);
         users.add (usuário);
se (usuários.tamanho() >= maxResults) quebrar;
      }
      usuários de retorno;
   }
```

O método getUsers() iteriza sobre o conjunto-chave do arquivo de propriedade, delegando-se a obterUserByUsername() para carregar um usuário. Observe que estamos indexando esta chamada com base no primeiro parâmetroResult e maxResults. Se sua loja externa não suportar paginação, você terá que fazer lógica semelhante.

PropertyFileUserStorageProvider

```
   @Override
Public List<UserModel> searchForUser(String search, RealmModel realm) {
pesquisa de retornoPara Usuário (pesquisa, reino, 0, Integer.MAX_VALUE);
   }

   @Override
lista pública<UserModel> pesquisaForUser(Pesquisa de string, realm, int firstResult, int
maxResults) {
Lista<UserModel> usuários = novo LinkedList<>();
int i = 0;
      para (Objeto obj : properties.keySet()) {
         Nome de usuário de string = (String)obj;
se (!username.contém(pesquisa)) continuar;
se (i++ < primeiroResult) continuar;
         UsuárioModel usuário = getUserByUsername (nome de usuário, reino);
```

```
            users.add (usuário);
se (usuários.tamanho() >= maxResults)  quebrar;
        }
        usuários de retorno;
    }
```

A primeira declaração de pesquisaParaUser() leva um parâmetro String. Este é suposto ser uma sequência que você usa para pesquisar atributos de nome de usuário e e-mail para encontrar o usuário. Esta sequência pode ser um substring, e é por isso que usamos o método String.contains() ao fazer nossa pesquisa.

PropertyFileUserStorageProvider

```
    @Override
Public List<UserModel> searchForUser (Map<String, String> params, RealmModel realm) {
busca de retornoParaUser(params, reino,  0, Integer.MAX_VALUE);
    }

    @Override
Public List<UserModel> searchForUser (Map<String, String> params, RealmModel realm,  int
firstResult,  int  maxResults) {
        apenas suporte de pesquisa por nome de usuário
        String usernameSearchString = params.get("nome de usuário");
se (nome de usuárioSearchString ==  nulo)  retornar  Collections.EMPTY_LIST;
pesquisa de retornoParaUser (nome de usuárioSearchString, reino, primeiroResult,
maxResults);
    }
```

O método searchForUser() que toma um parâmetro mapa pode procurar um usuário com base no primeiro, sobrenome, nome de usuário e e-mail. Nós só armazenamos nomes de usuário, então só pesquisamos com base em nomes de usuário. Nós delegados para procurar PorUser() para isso.

PropertyFileUserStorageProvider

```
    @Override
Lista pública<UserModel> getGroupMembers (RealmModel realm, GroupModel group,  int
firstResult,  int  maxResults) {
        Collections.EMPTY_LIST de retorno;
    }

    @Override
Lista pública<UserModel> getGroupMembers (RealmModel realm, GroupModel group) {
        Collections.EMPTY_LIST de retorno;
    }

    @Override
lista pública<UserModel> pesquisaForUserByUserAttribute (String attrName, String attrValue,
RealmModel realm) {
        Collections.EMPTY_LIST de retorno;
    }
```

Não armazenamos grupos ou atributos, então os outros métodos retornam uma lista vazia.

Aumentando o armazenamento externo

O exemplo do PropertyFileUserStorageProvider é realmente limitado. Embora possamos fazer login com usuários armazenados em um arquivo de propriedade, não seremos capazes de fazer muito mais. Se os usuários carregados por este provedor precisarem de mapeamentos de papéis ou grupos especiais para acessar totalmente aplicativos específicos, não há como adicionar mapeamentos adicionais de papéis a esses usuários. Você também não pode modificar ou adicionar atributos importantes adicionais, como e-mail, primeiro e sobrenome.

Para esses tipos de situações, o Keycloak permite que você aumente sua loja externa armazenando informações extras no banco de dados do Keycloak. Isso é chamado de armazenamento de usuário federado e é encapsulado dentro da classe org.keycloak.storage.federated.UserFederatedStorageProvider.

UserFederatedStorageProvider

```
pacote org.keycloak.storage.federado;

interface pública UsuárioFederatedStorageProvider estende Provedor {

   Set<GroupModel> getGroups (RealmModel realm, String userId);
   void joinGroup (RealmModel realm, String userId, GroupModel group);
   vazio deixagroup (Reinommodel, Usuário stringId, grupo GroupModel);
Lista<String> getMembership (RealmModel realm, GroupModel group, int firstResult, int
max);

...
```

A instância UserFederatedStorageProvider está disponível no método KeycloakSession.userFederatedStorage(). Possui todos os diferentes tipos de métodos para armazenar atributos, mapeamentos de grupos e papéis, diferentes tipos de credenciais e ações necessárias. Se o modelo de dados da sua loja externa não puder suportar o conjunto completo do recurso Keycloak, este serviço poderá preencher as lacunas.

O Keycloak vem com uma classe helper org.keycloak.storage.adapter.AbstractUserAdapterFederatedStorage que irá delegar todos os métodos do UserModel, exceto obter/definir o nome de usuário para o armazenamento federado pelo usuário. Anular os métodos que você precisa substituir para delegar às suas representações de armazenamento externo. É fortemente sugerido que você leia o javadoc desta classe, pois tem métodos protegidos menores que você pode querer substituir. Especificamente em torno de membros de grupos e mapeamentos de papéis.

Exemplo de aumento

Em nosso exemplo de PropertyFileUserStorageProvider, só precisamos de uma simples mudança em nosso provedor para usar o AbstractUserAdapterFederatedStorage.

PropertyFileUserStorageProvider

```
UserModel protegido criarAdapter (RealmModel realm, String username) {
      retornar novo AbstractUserAdapterFederatedStorage (sessão, reino, modelo) {
        @Override
        cadeia pública getUsername() {
          nome de usuário de retorno;
        }

        @Override
        conjunto de vazio públicoSa nome de usuário (nome de usuário string) {
          String pw = (String)properties.remove (nome de usuário);
se (pw != nulo){
              propriedades.put (nome de usuário, pw);
              salvar();
          }
        }
      };
  }
```

Em vez disso, definimos uma implementação de classe anônima do AbstractUserAdapterFederatedStorage. O método setUsername() faz alterações no arquivo de propriedades e o salva.

Estratégia de Implementação de Importação

Ao implementar um provedor de armazenamento de usuários, há outra estratégia que você pode tomar. Em vez de usar o armazenamento federado pelo usuário, você pode criar um usuário localmente no banco de dados de usuário incorporado keycloak e copiar atributos de sua loja externa para esta cópia local. Há muitas vantagens nessa abordagem.

- Keycloak basicamente se torna um cache de usuário de persistência para sua loja externa. Uma vez que o usuário é importado, você não vai mais bater na loja externa, assim tirando a carga dele.
- Se você estiver se mudando para keycloak como sua loja oficial de usuários e depreciando a antiga loja externa, você pode migrar lentamente aplicativos para usar o Keycloak. Quando todos os aplicativos tiverem sido migrados, desvincule o usuário importado e retire a antiga loja externa legado.

Há algumas desvantagens óbvias, porém, em usar uma estratégia de importação:

- Procurar um usuário pela primeira vez exigirá várias atualizações para o banco de dados Keycloak. Isso pode ser uma grande perda de desempenho sob carga e colocar muita pressão no banco de dados Keycloak. A abordagem de armazenamento federada pelo usuário só armazenará dados extras conforme necessário e pode nunca ser usada dependendo dos recursos da sua loja externa.
- Com a abordagem de importação, você tem que manter o armazenamento local keycloak e armazenamento externo em sincronia. O SPI de armazenamento do usuário tem interfaces de capacidade que você pode implementar para suportar a sincronização, mas isso pode rapidamente se tornar doloroso e confuso.

Para implementar a estratégia de importação basta verificar se o usuário foi importado localmente. Se assim for, devolva o usuário local, se não criar o usuário localmente e importar dados da loja externa. Você também pode proxy do usuário local para que a maioria das alterações sejam sincronizadas automaticamente.

Isso será um pouco inventado, mas podemos estender nosso `PropertyFileUserStorageProvider` para adotar essa abordagem. Começamos primeiro modificando o método `createAdapter()`.

PropertyFileUserStorageProvider

```
UserModel protegido criarAdapter (RealmModel realm, String username) {
      UserModel local = session.userLocalStorage().getUserByUsername(nome de usuário,
reino);
se (local ==  nulo){
         local = session.userLocalStorage().addUser(reino, nome de usuário);
         local.setFederationLink (model.getId());
      }
      retornar novo UserModelDelegate (local) {
         @Override
         conjunto de vazio públicoSa nome de usuário (nome de usuário string) {
            String pw = (String)properties.remove (nome de usuário);
se (pw !=  nulo){
               propriedades.put (nome de usuário, pw);
               salvar();
            }
            super.setUsername (nome de usuário);
         }
      };
   }
```

Neste método chamamos o método `KeycloakSession.userLocalStorage()` para obter uma referência ao armazenamento local do usuário Keycloak. Vemos se o usuário é armazenado localmente, se não, nós adicionamos localmente. Não defina a identificação do usuário local. Deixe keycloak gerar automaticamente o `id`. Observe também que chamamos `usermodel.setFederationLink()` e passamos no ID do

ComponentModel do nosso provedor. Isso define um link entre o provedor e o usuário importado.

> Quando um provedor de armazenamento de usuário for removido, qualquer usuário importado por ele também será removido. Este é um dos propósitos de chamar UserModel.setFederationLink().

Outra coisa a notar é que, se um usuário local estiver vinculado, seu provedor de armazenamento ainda será delegado para métodos que implementa a partir das interfaces CredentialInputValidator e CredentialInputUpdater. O retorno falso de uma validação ou atualização resultará apenas em Keycloak vendo se ele pode validar ou atualizar usando o armazenamento local.

Observe também que estamos proxying o usuário local usando a classe org.keycloak.models.utils.UserModelDelegate. Esta classe é uma implementação do UserModel. Todos os métodos apenas delegam ao UserModel com o qual foi instanciado. Nós substituimos o método setUsername() desta classe de delegado para sincronizar automaticamente com o arquivo de propriedade. Para seus provedores, você pode usá-lo para *interceptar* outros métodos no UserModel local para realizar a sincronização com sua loja externa. Por exemplo, obter métodos poderia garantir que a loja local está em sincronia. Definir métodos manter a loja externa em sincronia com a local. Uma coisa a notar é que o método getId() deve sempre retornar o id que foi gerado automaticamente quando você criou o usuário localmente. Você não deve devolver um id federado como mostrado nos outros exemplos de não importação.

> Se o seu provedor estiver implementando a interface UserRegistrationProvider, seu método removeUser() não precisará remover o usuário do armazenamento local. O tempo de execução executará automaticamente esta operação. Observe também que remover Usuário() será invocado antes de ser removido do armazenamento local.

Interface de avaliação de usuário importado

Se você se lembra no início deste capítulo, discutimos como funcionava a consulta para um usuário. O armazenamento local é consultado primeiro, se o usuário for encontrado lá, então a consulta termina. Este é um problema para a nossa implementação acima, pois queremos proxy do UserModel local para que possamos manter os nomes de usuário em sincronia. O SPI de armazenamento do usuário tem um retorno de chamada para sempre que um usuário local vinculado é carregado do banco de dados local.

```
pacote org.keycloak.storage.user;
interface pública ImportedUserValidation {
  /**
   * Se este método retornar nulo, então o usuário no armazenamento local será removido
   *
   @param reino
```

```
     * @param usuário
     * @return nulo se o usuário não for mais válido
     */
    UserModel valida (RealmModel realm, UserModel user);
}
```

Sempre que um usuário local vinculado é carregado, se a classe do provedor de armazenamento do usuário implementar essa interface, então o método `valid ()` é chamado. Aqui você pode proxy o usuário local passou como um parâmetro e devolvê-lo. Esse novo `Modelo de Usuário` será usado. Você também pode fazer uma verificação opcional para ver se o usuário ainda existe na loja externa. Se `validar()` retorna `nulo`,então o usuário local será removido do banco de`dados`.

Interface de importaçãoSincronização

Com a estratégia de importação, você pode ver que é possível que a cópia do usuário local saia da sincronia com o armazenamento externo. Por exemplo, talvez um usuário tenha sido removido da loja externa. O SPI de armazenamento do usuário tem uma interface adicional que você pode implementar para lidar com isso, `org.keycloak.storage.user.ImportSynchronization`:

```
pacote org.keycloak.storage.user;

interface pública ImportaçãoSsincronização  {
    SincronizaçãoSult (KeycloakSessionFactory sessionFactory, String realmId,
UserStorageProviderModel);
    SincronizaçãoResult syncSince (Data últimaSync, sessão KeycloakSessionFactoryFactory,
String realmId, UserStorageProviderModel);
}
```

Esta interface é implementada pela fábrica do provedor. Uma vez que esta interface é implementada pela fábrica do provedor, a página de gerenciamento do console de administração para o provedor mostra opções adicionais. Você pode forçar manualmente uma sincronização clicando em um botão. Isso invoca o método `ImportSynchronization.sync()`. Além disso, opções adicionais de configuração são exibidas que permitem agendar automaticamente uma sincronização. Sincronizações automáticas invocam o método `syncSince()`.

Caches de usuários

Quando um objeto de usuário é carregado por consultas de ID, nome de usuário ou e-mail, ele é armazenado em cache. Quando um objeto do usuário está sendo armazenado em cache, ele itera através de toda a interface `UserModel` e puxa essas informações para um cache local somente na memória. Em um cluster, este cache ainda é local, mas se torna um cache de invalidação. Quando um objeto do usuário é

modificado, ele é despejado. Este evento de despejo é propagado para todo o cluster para que o cache do usuário dos outros nós também seja invalidado.

Gerenciamento do cache do usuário

Você pode acessar o cache do usuário ligando para `KeycloakSession.userCache()`.

```
/**
 * Todos esses métodos afetam um conjunto inteiro de instâncias Keycloak.
 *
 * @author <a href="mailto:bill@burkecentral.com">Bill Burke</a>
 * @version $Revision: 1 $
 */
interface pública UserCache  estende  UserProvider {
   /**
    * Usuário de despejo de cache.
    *
    * @param usuário
    */
despejo vazio (Reinommodel, usuário do UserModel);

   /**
    * Usuários de despejo de um reino específico
    *
    @param reino
    */
despejo vazio (Reinomodel);

   /**
    * Limpar totalmente o cache.
    *
    */
   vazio claro();
}
```

Existem métodos para despejar usuários específicos, usuários contidos em um reino específico ou em todo o cache.

OnUserCache Callback Interface

Você pode querer armazenar informações adicionais específicas para a implementação do seu provedor. O SPI de armazenamento do usuário tem um retorno de chamada sempre que um usuário é armazenado em cache: `org.keycloak.models.cache.OnUserCache`.

```
interface pública OnUserCache  {
   vazio onCache (realmmodel realm, usuário CachedUserModel, usuário do UserModel);
}
```

Sua classe de provedor deve implementar essa interface se quiser esse retorno de chamada. O parâmetro de delegado UserModel é a instância Do UsuárioModel devolvida pelo seu provedor. O CachedUserModel é uma interface expandida do UserModel. Este é o exemplo que é armazenado localmente no armazenamento local.

```
interface pública CachedUserModel estende UserModel {

  /**
   * Invalida o cache para este usuário e devolve um delegado que representa o provedor de dados real
   *
   @return
   */
  UserModel getDelegateForUpdate();

  boolean isMarkedForEviction();

  /**
   * Invalidar o cache para este modelo
   *
   */
  nulidade do vazio();

  /**
   * Quando foi o modelo foi carregado do banco de dados.
   *
   @return
   */
getCacheTimestamp();

  /**
   * Retorna um mapa que contém coisas personalizadas que são armazenadas em cache junto com este modelo.  Você pode escrever para este mapa.
   *
   @return
   */
  ConcurrentHashMap getCachedWith();
}
```

Esta interface CachedUserModel permite que você despeje o usuário do cache e obtenha a instância do UserModel do provedor. O método getCachedWith() retorna um mapa que permite que você em cache de informações adicionais relativas ao usuário. Por exemplo, as credenciais não fazem parte da interface UserModel. Se você quisesse armazenar credenciais em cache na memória, você implementaria o OnUserCache e armazenaria as credenciais do usuário usando o método getCachedWith().

Políticas de cache

Na página de gerenciamento do console de administração para o seu provedor de armazenamento de usuários, você pode especificar uma política de cache exclusiva.

Alavancando Java EE

Os provedores de armazenamento de usuários podem ser embalados em qualquer componente Java EE se você configurar o arquivo `META-INF/serviços` corretamente para apontar para seus provedores. Por exemplo, se o seu provedor precisar usar bibliotecas de terceiros, você pode empacotar seu provedor dentro de um EAR e armazenar essas bibliotecas de terceiros no `diretório lib/do` EAR. Observe também que os JARs do provedor podem fazer uso da estrutura de implantação `jboss.xml` arquivo que EJBs, WARS e EARs podem usar em um ambiente WildFly. Para obter mais detalhes sobre este arquivo, consulte a documentação do WildFly. Ele permite que você puxe em dependências externas entre outras ações de grãos finos.

As implementações do provedor são necessárias para serem objetos java simples. Mas também apoiamos a implementação das classes `UserStorageProvider` como EJBs stateful. Isso é especialmente útil se você quiser usar JPA para se conectar a uma loja relacional. É assim que você faria:

```
@Stateful
@Local(EjbExampleUserStorageProvider.class)
classe pública EjbExampleUserStorageProvider  implementa  UserStorageProvider,
      UserLookupProvider,
      UserRegistrationProvider,
      UserQueryProvider,
      CredencialInputUpdater,
      CredencialInputValidator,
      OnUserCache
{
   @PersistenceContext
Entidades protegidasManager em;

modelo de modelo de modelo de componente protegido;
sessão de sessão de sessão de keycloaksession protegida;

   conjunto de vazio públicoModel (modelo ComponentModel) {
      estemodelo .= modelo;
   }

   conjunto de vazio públicoSession (sessão KeycloakSession) {
      estasessão .session = sessão;
   }


   @Remove
   @Override
   vazio público perto() {
   }
...
}
```

Você tem que definir a `@Local` anotação e especificar sua classe de provedor lá. Se você não fizer isso, o EJB não fará a proxy do usuário corretamente e seu provedor não funcionará.

Você deve colocar a `anotação @Remove` no método `próximo ()` do seu provedor. Se você não fizer isso, o feijão imponente nunca será limpo e você pode eventualmente ver mensagens de erro.

As implementações do `UserStorageProvider` são necessárias para serem objetos Java simples. Sua classe de fábrica realizaria uma busca JNDI do EJB Stateful em seu método de criação().

```
classe pública EjbExampleUserStorageProviderFactory
      implementa UserStorageProviderFactory<EjbExampleUserStorageProvider> {

   @Override
ejbexampleUserStorageProvider create (KeycloakSession session, ComponentModel model) {
      tente {
InicialContext ctx = novo  Texto Inicial();
         EjbExampleUserStorageProvider provider =
(EjbExampleUserStorageProvider)ctx.lookup(
               "java:global/user-storage-jpa-example/" +
EjbExampleUserStorageProvider.class.getSimpleName());
         provider.setModel(modelo);
         provider.setSession(sessão);
provedor de retorno;
} captura  (Exceção e) {
         lançar novo RuntimeException(e);
      }
   }
```

Este exemplo também assume que você definiu uma implantação JPA no mesmo JAR que o provedor. Isso significa um arquivo `.xml persistência`, bem como qualquer classe `jpa @Entity`.

> Ao usar JPA, qualquer fonte de dados adicional deve ser uma fonte de dados XA. A fonte de dados Keycloak não é uma fonte de dados XA. Se você interagir com duas ou mais fontes de dados não-XA na mesma transação, o servidor retorna uma mensagem de erro. Apenas um recurso não-XA é permitido em uma única transação. Consulte o manual wildfly para obter mais detalhes sobre a implantação de uma fonte de dados XA.

O CDI não é suportado.

API de gerenciamento de descanso

Você pode criar, remover e atualizar as implantações do provedor de armazenamento de usuários através da API REST do administrador. O SPI de armazenamento do usuário é construído em cima de uma interface de componente genérico para que você use essa API genérica para gerenciar seus provedores.

A API do componente REST vive sob o recurso de administração do seu reino.

```
/administrador/reinos/{nome de reino}/componentes
```

Só mostraremos essa interação de API REST com o cliente Java. Espero que você possa extrair como fazer isso a partir de `curl` a partir desta API.

```
interface pública ComponentsResource {
   @GET
   @Produces(MediaType.APPLICATION_JSON)
consulta <relação de acordos de trabalho pública>);

   @GET
   @Produces(MediaType.APPLICATION_JSON)
consulta pública List<ComponentRepresentation> (@QueryParam(pai)String parent);

   @GET
   @Produces(MediaType.APPLICATION_JSON)
lista pública<Representação de especialistas> consulta(@QueryParam(pai)Pai de corda,
@QueryParam("tipo") Tipo de string);

   @GET
   @Produces(MediaType.APPLICATION_JSON)
consulta pública List<ComponentRepresentation> (@QueryParam("pai") String parent,
@QueryParam(tipo") Tipo de string,
@QueryParam("nome") Nome da string);

   @POST
   @Consumes(MediaType.APPLICATION_JSON)
   Adicionar resposta (representante de representação de representação de componentes);

@Path ("{id}")
ComponenteResource(@PathParam("id") ID de corda);
}

interface pública ComponentResource {
   @GET
   representação de componentes públicos para representação();

   @PUT
   @Consumes(MediaType.APPLICATION_JSON)
   Atualização de vazio público (Representante de Representação de Componentes);

   @DELETE
   vazio público remover();
}
```

Para criar um provedor de armazenamento de usuário, você deve especificar o id do provedor, um tipo de provedor do string `org.keycloak.storage.UserStorageProvider`,bem como a configuração.

```
import org.keycloak.admin.client.Keycloak;
import org.keycloak.representações.idm.RealmRepresentation;
...

Keycloak keycloak = Keycloak.getInstance(
"http://localhost:8080/auth",
"mestre",
"administrador",
"senha",
   "admin-cli");
ReinomResource realmResource = keycloak.realm("mestre");
Reino derepresentação de reino = realmResource.toRepresentation();

ComponenteRepresentação de componentes = nova  Representação de Componentes();
component.setName("home");
component.setProviderId("readonly-property-file");
component.setProviderType("org.keycloak.storage.UserStorageProvider");
componente.setParentId(realm.getId());
component.setConfig(novo  MultivalordHashMap());
componente.getConfig().putSingle("path",  "~/users.properties");

reinoResource.components().add(componente);

recuperar um componente

List<ComponentRepresentation> componentes =
realmResource.components().query(realm.getId(),
"org.keycloak.storage.UserStorageProvider",
"casa");
componente = componentes.get(0);

Atualize um componente

component.getConfig().putSingle("path",  "~/my-users.properties");
realmResource.components().component(component.getId()).update(component);

Remova um componente

realmREsource.components().component(component.getId()).remove();
```

Migrando de um SPI da Federação de Usuários Anterior

> Este capítulo só é aplicável se você tiver implementado um provedor usando o SPI anterior (e agora removido) da Federação do Usuário.

Na versão 2.4.0 do Keycloak e anteriormente havia um SPI da Federação do Usuário. Red Hat Single Sign-On versão 7.0, embora sem suporte, tinha este SPI anterior disponível também. Este SPI anterior da Federação do Usuário foi removido da versão 2.5.0 do Keycloak e da versão 7.1 do Red Hat Single Sign-On. No entanto, se você escreveu um provedor com este SPI anterior, este capítulo discute algumas estratégias que você pode usar para portar.

Importação vs. Não-Importação

O SPI anterior da Federação do Usuário exigia que você criasse uma cópia local de um usuário no banco de dados do Keycloak e importasse informações de sua loja externa para a cópia local. No entanto, isso não é mais um requisito. Você ainda pode portar seu provedor anterior como está, mas você deve considerar se uma estratégia de não importação pode ser uma abordagem melhor.

Vantagens da estratégia de importação:

- Keycloak basicamente se torna um cache de usuário de persistência para sua loja externa. Uma vez que o usuário é importado, você não vai mais chegar à loja externa, tirando assim a carga dele.
- Se você estiver se mudando para keycloak como sua loja oficial de usuários e depreciando a loja externa anterior, você pode migrar lentamente aplicativos para usar o Keycloak. Quando todos os aplicativos tiverem sido migrados, desvincule o usuário importado e retire a loja externa anteriormente legado.

Há algumas desvantagens óbvias, porém, em usar uma estratégia de importação:

- Procurar um usuário pela primeira vez exigirá várias atualizações para o banco de dados Keycloak. Isso pode ser uma grande perda de desempenho sob carga e colocar muita pressão no banco de dados Keycloak. A abordagem de armazenamento federada pelo usuário só armazenará dados extras conforme necessário e nunca poderá ser usada dependendo dos recursos da sua loja externa.
- Com a abordagem de importação, você tem que manter o armazenamento local keycloak e armazenamento externo em sincronia. O SPI de armazenamento do usuário tem interfaces de capacidade que você pode implementar para suportar a sincronização, mas isso pode rapidamente se tornar doloroso e confuso.

UserFederationProvider vs. UserStorageProvider

A primeira coisa a notar é que o `UserFederationProvider` era uma interface completa. Você implementou todos os métodos nesta interface. No entanto, o `UserStorageProvider`, em vez disso, dividiu essa interface em várias interfaces de capacidade que você implementa conforme necessário.

UserFederationProvider.getUserByUsername() e getUserByEmail() têm equivalentes exatos no novo SPI. A diferença entre os dois é como você importa. Se você vai continuar com uma estratégia de importação, você não chama mais o KeycloakSession.userStorage().addUser() para criar o usuário localmente. Em vez disso, você chama KeycloakSession.userLocalStorage().addUser(). O método userStorage() não existe mais.

O método UserFederationProvider.validateAndProxy() foi movido para uma interface de capacidade opcional, ImportedUserValidation. Você deseja implementar esta interface se estiver portando seu provedor anterior como está. Observe também que no SPI anterior, esse método era chamado toda vez que o usuário era acessado, mesmo que o usuário local estivesse no cache. No SPI posterior, este método só é chamado quando o usuário local é carregado do armazenamento local. Se o usuário local estiver em cache, o método ImportedUserValidation.validate() não será chamado de forma alguma.

O método UserFederationProvider.isValid() não existe mais no SPI posterior.

Os métodos UserFederationProvider sincronizamRegistrações(), registramUser()e removemUser() foram movidos para a interface de capacidade UserRegistrationProvider. Esta nova interface é opcional de implementação, portanto, se o seu provedor não suporta criar e remover usuários, você não precisa implementá-la. Se o seu provedor anterior tiver alternado para alternar o suporte para registrar novos usuários, isso será suportado no novo SPI, retornando nulo do UserRegistrationProvider.addUser() se o provedor não suportar a adição de usuários.

Os métodos anteriores do UserFederationProvider centrados em credenciais estão agora encapsulados nas interfaces CredencialInputValidator e CredentialInputUpdater, que também são opcionais de implementar, dependendo se você suporta validar ou atualizar credenciais. O gerenciamento de credenciais costumava existir nos métodos UserModel. Estes também foram movidos para as interfaces CredentialInputValidator e CredentialInputUpdater. Uma coisa é notar que se você não implementar a interface CredentialInputUpdater, então quaisquer credenciais fornecidas pelo seu provedor podem ser substituídas localmente no armazenamento Keycloak. Então, se você quiser que suas credenciais sejam somente leitura, implemente o método CredencialInputUpdater.updateCredential() e retorne um ReadOnlyException.

Os métodos de consulta UserFederationProvider, como o searchByAttributes() e o getGroupMembers() estão agora encapsulados em uma interface opcional UserQueryProvider. Se você não implementar esta interface, então os usuários não serão visualizados no console administrativo. Você ainda poderá fazer login.

UserFederationProviderFactory vs. UserStorageProviderFactory

Os métodos de sincronização no SPI anterior estão agora encapsulados dentro de uma interface opcional de isnchronização de importação. Se você implementou a lógica de

sincronização, então faça com que seu novo UserStorageProviderFactory implemente a interface ImportSynchronization.

Atualização para um novo modelo

As instâncias SPI de armazenamento do usuário são armazenadas em um conjunto diferente de tabelas relacionais. Keycloak executa automaticamente um script de migração. Se algum provedor anterior da Federação de Usuários for implantado para um reino, eles ão convertidos para o modelo de armazenamento posterior como está, incluindo a identificação dos dados. Essa migração só acontecerá se um provedor de armazenamento de usuário existir com o mesmo ID do provedor (ou seja, "ldap", "kerberos") como o provedor anterior da Federação de Usuários.

Então, sabendo disso, existem diferentes abordagens que você pode tomar.

1. Você pode remover o provedor anterior em sua implantação keycloak anterior. Isso removerá as cópias vinculadas locais de todos os usuários importados. Em seguida, ao atualizar o Keycloak, basta implantar e configurar seu novo provedor para o seu reino.
2. A segunda opção é escrever seu novo provedor certificando-se de que ele tenha o mesmo ID do provedor: UserStorageProviderFactory.getId(). Certifique-se de que este provedor está no diretório autônomo/implantações/diretório da nova instalação Keycloak. Inicializar o servidor e ter o script de migração incorporado convertido do modelo de dados anterior para o modelo de dados posterior. Neste caso, todos os seus usuários importados vinculados anteriormente funcionarão e serão os mesmos.

Se você decidiu se livrar da estratégia de importação e reescrever seu provedor de armazenamento de usuários, sugerimos que você remova o provedor anterior antes de atualizar o Keycloak. Isso removerá cópias importadas locais vinculadas de qualquer usuário importado.

Interfaces baseadas em fluxo

Muitas das interfaces de armazenamento do usuário no Keycloak contêm métodos de consulta que podem retornar conjuntos potencialmente grandes de objetos, o que pode levar a impactos significativos em termos de consumo de memória e tempo de processamento. Isso é especialmente verdade quando apenas um pequeno subconjunto do estado interno dos objetos é usado na lógica do método de consulta.

Para fornecer aos desenvolvedores uma alternativa mais eficiente para processar grandes conjuntos de dados nesses métodos de consulta, uma sub-interface do Streams foi adicionada às interfaces de armazenamento do usuário. Essas sub-interfaces do Streams substituem os métodos originais baseados em coleção nas super-interfaces

com variantes baseadas em fluxo, tornando os métodos baseados em coleta padrão. A implementação padrão de um método de consulta baseado em coleta invoca sua contraparte Do Fluxo e coleta o resultado no tipo de coleta adequado.

As subco interagens do Streams permitem que as implementações se concentrem na abordagem baseada em fluxo para processar conjuntos de dados e se beneficiem das otimizações potenciais de memória e desempenho dessa abordagem. As interfaces que oferecem uma sub-interface do Streams a ser implementada incluem algumas *interfaces de capacidade,* todas as interfaces no pacote org.keycloak.storage.federated e algumas outras que podem ser implementadas dependendo do escopo da implementação personalizada do armazenamento.

Veja esta lista das interfaces que oferecem uma sub-interface do Streams para desenvolvedores.

pacote	Classes
org.keycloak.credencial	CredencialInputUpdater(*), UserCredentialStore
org.keycloak.models	GroupModel, RoleMapperModel, UserCredentialManager, UserModel, UserProvider
org.keycloak.models.cache	CachedUserModel, UserCache
org.keycloak.storage.federado	Todas as interfaces
org.kecyloak.storage.user	UserQueryProvider(*)

(*) indica que a interface é uma *interface de capacidade*

A implementação personalizada de armazenamento do usuário que deseja se beneficiar da abordagem de fluxos deve simplesmente implementar as sub-interfaces do Streams em vez das interfaces originais. Por exemplo, o código a seguir usa a variante Streams da interface UserQueryProvider:

```
classe pública CustomQueryProvider estende UserQueryProvider.Streams {
...
  @Override
  Stream<UserModel> getUsersStream (RealmModel realm, Integer firstResult, Integer
maxResults) {
    lógica personalizada aqui
  }

  @Override
  Stream<UserModel> pesquisaForUserStream (Pesquisa de string, reino realmmodel) {
    lógica personalizada aqui
  }
```

```
...
}
```

SPI do cofre

Provedor de cofre

Você pode usar um SPI de cofre do pacote `org.keycloak.vault` para escrever extensão personalizada para keycloak para se conectar à implementação arbitrária do cofre.

O provedor de `textos embutidos` é um exemplo da implementação deste SPI. Em geral, aplicam-se as seguintes regras:

- Para evitar que um segredo vaze através de reinos, você pode querer isolar ou limitar os segredos que podem ser recuperados por um reino. Nesse caso, seu provedor deve levar em conta o nome do reino ao procurar segredos, por exemplo, prefixando entradas com o nome do reino. Por exemplo, uma expressão `${vault.key}` avaliaria geralmente para diferentes nomes de entrada, dependendo se foi usado em um reino *A* ou reino *B*. Para diferenciar entre os reinos, o reino precisa ser passado para a instância criada do `VaultProvider` do método `VaultProviderFactory.create()`, onde está disponível no parâmetro `KeycloakSession`.
- O provedor de cofre precisa implementar um único método que `obtenhaSecret` que devolva um `VaultRawSecret` pelo nome secreto dado. Essa classe detém a representação do segredo em `byte[]` ou `ByteBuffer` e espera-se que se converta entre os dois sob demanda. Observe que este buffer seria descartado após o uso, conforme explicado abaixo.

Em relação à separação de reinos, todas as fábricas de provedores de cofre incorporados permitem a configuração de um ou mais resolvedores-chave. Representado pela interface `VaultKeyResolver`, um resolvedor-chave essencialmente implementa o algoritmo ou estratégia para combinar o nome do reino com a chave (como obtido da expressão `${vault.key}` no nome de entrada final que será usado para recuperar o segredo do cofre. O código que lida com essa configuração foi extraído em classes de fábrica de provedor de cofre abstrato e provedor de cofre, de modo que implementações personalizadas que desejam oferecer suporte para resolvers-chave podem estender essas classes abstratas em vez da implementação de interfaces SPI para herdar a capacidade de configurar os principais resolvedores que devem ser tentados ao recuperar um segredo.

Para obter detalhes sobre como empacotar e implantar um provedor personalizado, consulte o capítulo Interfaces do Provedor de Serviços.

Consumindo valores do cofre

O cofre contém dados confidenciais e Keycloak trata os segredos de acordo. Ao acessar um segredo, o segredo é obtido do cofre e retido na memória JVM apenas pelo tempo necessário. Em seguida, todas as tentativas possíveis de descartar seu conteúdo da memória JVM são feitas. Isso é conseguido usando os segredos do cofre apenas dentro da instrução try-with-resources, conforme descrito abaixo:

```
char[] c;
try (VaultCharSecret cSecret = session.vault().getCharSecret(SECRET_NAME)) {
   // ... usar cSecret
   c = cSecret.getAsArray().orElse(nulo);
   se c != nulo, agora contém senha
}

se c != nulo, agora contém lixo
```

O exemplo usa KeycloakSession.vault() como ponto de entrada para acessar os segredos. O uso do método VaultProvider.obtainSecret diretamente também é possível. No entanto, o método vault() tem o benefício da capacidade de interpretar o segredo bruto (que geralmente é uma matriz de byte) como uma matriz de caracteres (via vault().getCharSecret()ouuma String (via vault().getStringSecret()além de obter o valor original não interpretado(via vault().getRawSecret().

Observe que, uma vez que os objetos string são imutáveis, seu conteúdo não pode ser descartado substituindo com lixo aleatório. Embora medidas tenham sido tomadas na implementação padrão do VaultStringSecret para evitar a internalização de Strings, os segredos armazenados em objetos String viveriam pelo menos até a próxima rodada GC. Assim, é preferível o uso de matrizes e buffers de caracteres simples e de caracteres.