Extending KEYCLOAK

for All Your Identity Use Cases

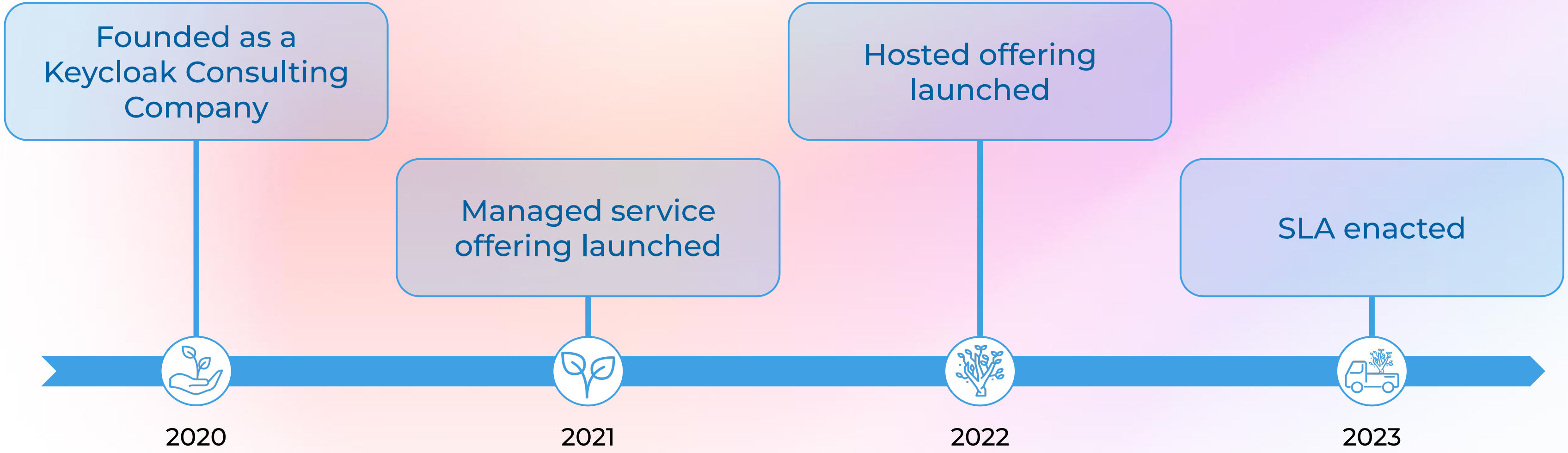
- *https://github.com/p2-inc*
- *Keycloak extensions*
- *Open source*

phase//

Who is Phase Two?

Keycloak extension, support and hosting.

Accelerate time-to-market and enterprise adoption for modern SaaS use cases.



Founded as a Keycloak Consulting Company

2020

Managed service offering launched

2021

Hosted offering launched

2022

SLA enacted

2023

Who am I?

i. *Serial entrepreneur (since 1996)*

ii. *Java user since 1.1 (1997)*

iii. *Keycloak user since just before v3 (2017)*

iv. *Founder/CEO Phase Two (2020)*

BrightRoll
SMART VIDEO ADVERTISING

4INFO

groupme

YAHOO!

What? Why?

- Keycloak is very mature, and handles 80% of use cases really well.
- For everything else, it is built as a set of Service Provider Interfaces (SPIs) and implementations that allow excellent configuration and programmatic extensibility
- **Extensions are the recommended mechanism to achieve custom functionality** that will not (or not soon) make it into Keycloak. “Custom” can mean:
  - Highly specific to me / my company
  - Something that may have broad appeal

But...

KC-SERVICES0047: events (io.phasetwo.keycloak.resources.WebhooksResourceProviderFactory) **is implementing the internal SPI realm-restapi-extension. This SPI is internal and may change without notice**

It depends…

This hasn’t been an issue, outside of UI and some other minor interface changes…

- But it means **you do need to validate each new version**
- Decide on your tolerance for risk based on **HOW MUCH KEYCLOAK USES IT INTERNALLY**
  - E.g. Authenticator won't change because they use it everywhere to implement flows
  - E.g. Admin UI extensions might change, because they don't use it to build the admin UI

phase//

Audience and Scope

Level: **Beginner**

Meant to answer the questions:

“I need to add to or extend Keycloak.
What is available? How do I get started?”

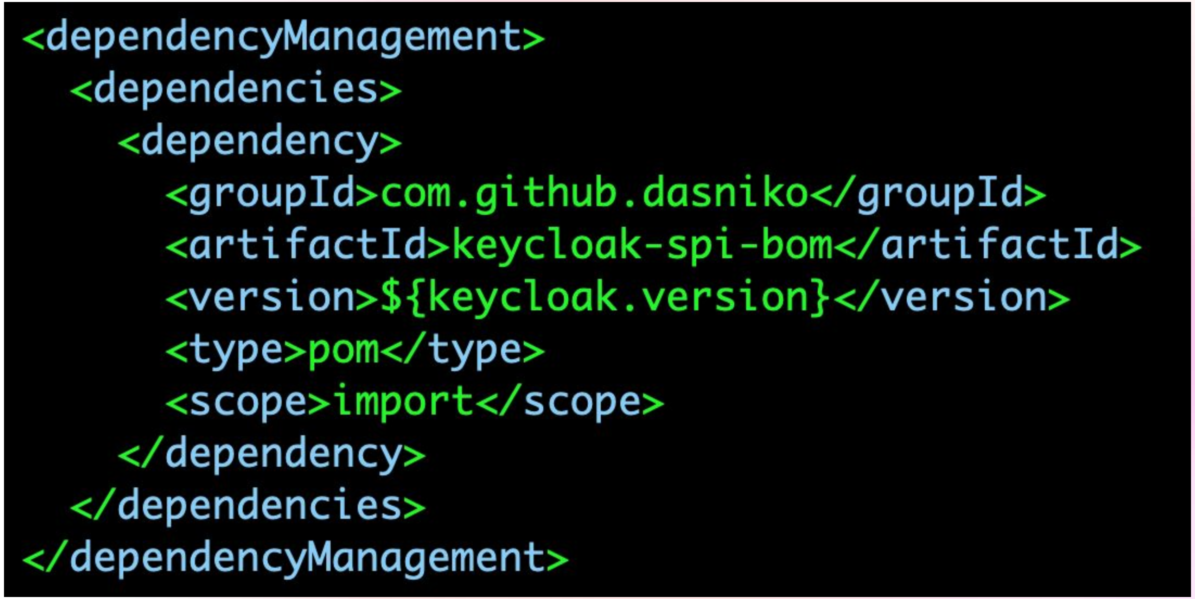
Sections:

1. Build setup
2. Lifecycle of an extension
3. Configuration
4. Testing
5. Live example

Build setup

1. The pom.xml file
   a. Use a BOM
   b. Annotation processors (lombok and auto-service)
   c. Adding version information to the ServerInfoAwareProviderFactory
   d. Testcontainers
   e. Building a fat jar

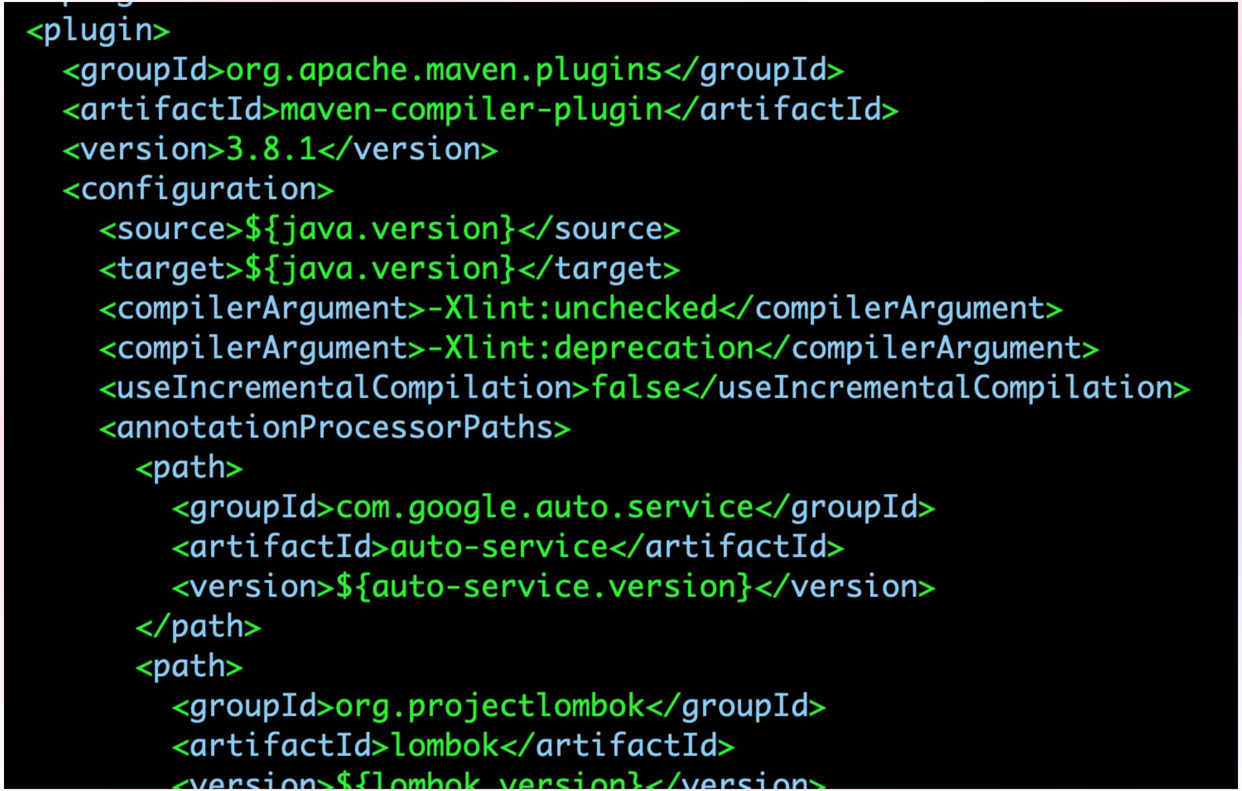
Build setup: Use a BOM



```
<dependencyManagement>
  <dependencies>
    <dependency>
      <groupId>com.github.dasniko</groupId>
      <artifactId>keycloak-spi-bom</artifactId>
      <version>${keycloak.version}</version>
      <type>pom</type>
      <scope>import</scope>
    </dependency>
  </dependencies>
</dependencyManagement>
```

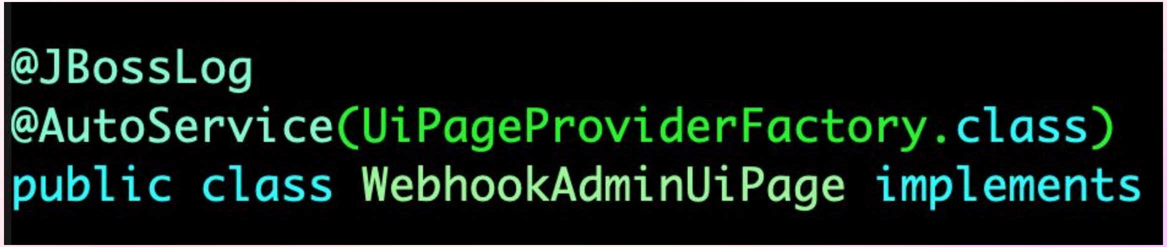
- Gives us versions and dependencies that match the Keycloak version
- No need to specify versions
- Thanks Niko!

Build setup: Annotation processors



```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-compiler-plugin</artifactId>
  <version>3.8.1</version>
  <configuration>
    <source>${java.version}</source>
    <target>${java.version}</target>
    <compilerArgument>-Xlint:unchecked</compilerArgument>
    <compilerArgument>-Xlint:deprecation</compilerArgument>
    <useIncrementalCompilation>false</useIncrementalCompilation>
    <annotationProcessorPaths>
      <path>
        <groupId>com.google.auto.service</groupId>
        <artifactId>auto-service</artifactId>
        <version>${auto-service.version}</version>
      </path>
      <path>
        <groupId>org.projectlombok</groupId>
        <artifactId>lombok</artifactId>
        <version>${lombok.version}</version>
```

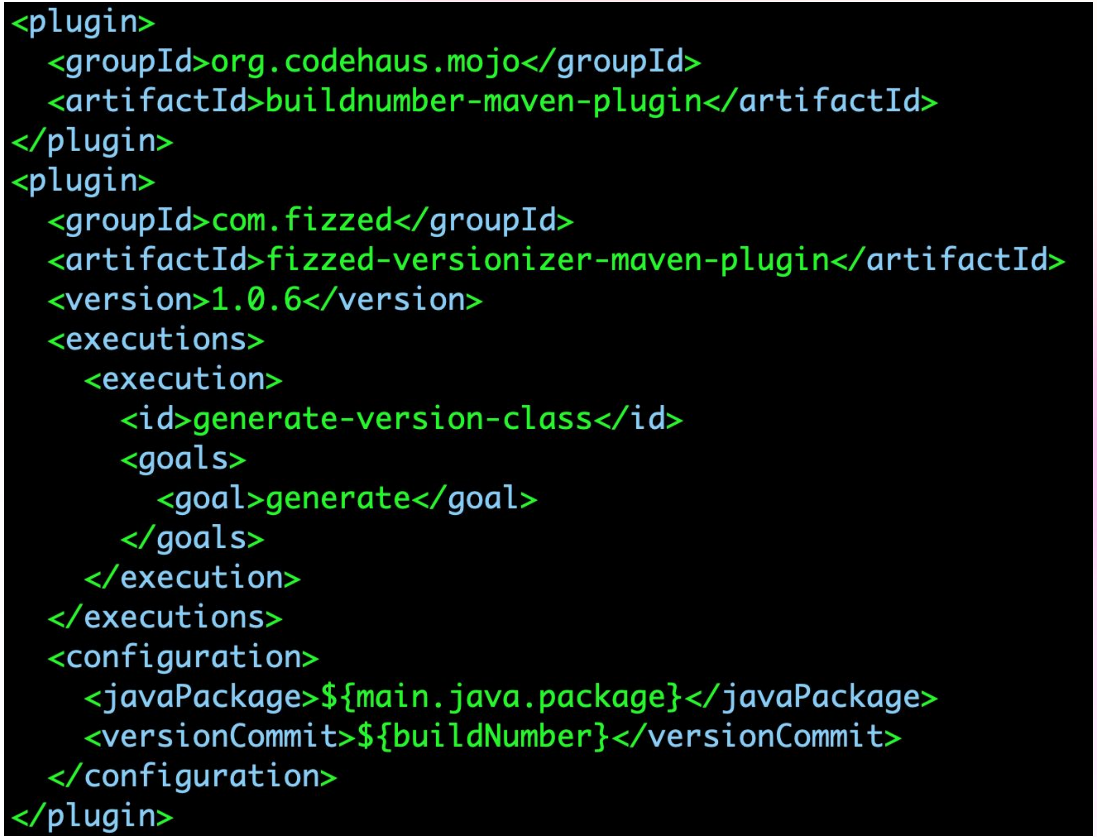
Build setup: Annotation processors



```
@JBossLog
@AutoService(UiPageProviderFactory.class)
public class WebhookAdminUiPage implements
```

- Auto-service
  - SPI manifest files are automatic
- Lombok
  - No more setting up logger for each class
  - Lots of other goodies for your data/record classes

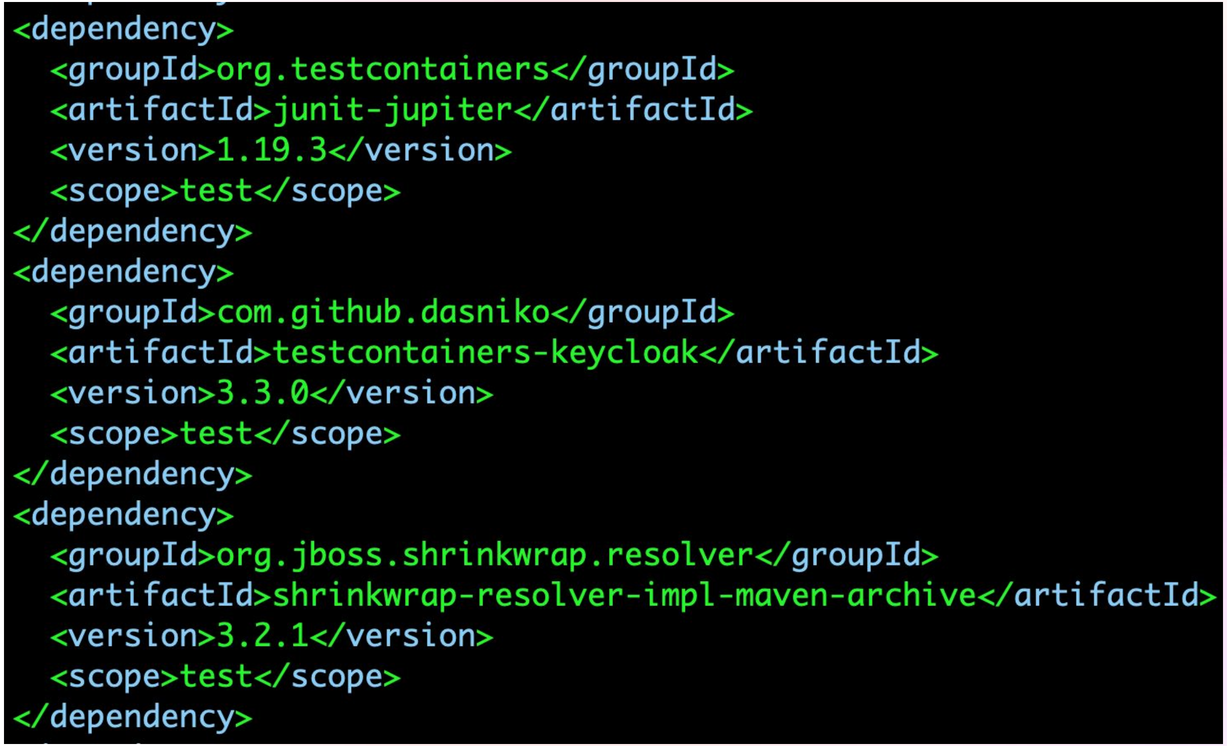
Build setup: Add metadata



```
<plugin>
  <groupId>org.codehaus.mojo</groupId>
  <artifactId>buildnumber-maven-plugin</artifactId>
</plugin>
<plugin>
  <groupId>com.fizzed</groupId>
  <artifactId>fizzed-versionizer-maven-plugin</artifactId>
  <version>1.0.6</version>
  <executions>
    <execution>
      <id>generate-version-class</id>
      <goals>
        <goal>generate</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <javaPackage>${main.java.package}</javaPackage>
    <versionCommit>${buildNumber}</versionCommit>
  </configuration>
</plugin>
```

- Adds project, version, and git sha to a generated class
- Easy to use when building your ServerInfoAwareProviderFactory

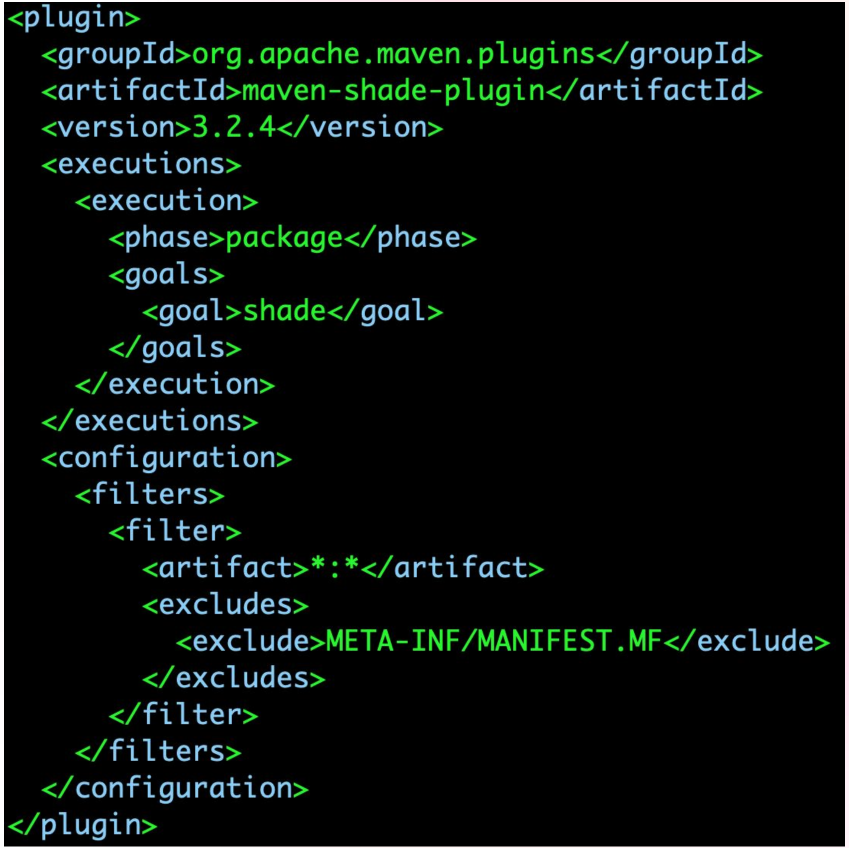
Build setup: Testcontainers



```
<dependency>
  <groupId>org.testcontainers</groupId>
  <artifactId>junit-jupiter</artifactId>
  <version>1.19.3</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>com.github.dasniko</groupId>
  <artifactId>testcontainers-keycloak</artifactId>
  <version>3.3.0</version>
  <scope>test</scope>
</dependency>
<dependency>
  <groupId>org.jboss.shrinkwrap.resolver</groupId>
  <artifactId>shrinkwrap-resolver-impl-maven-archive</artifactId>
  <version>3.2.1</version>
  <scope>test</scope>
</dependency>
```

- Testcontainers implementation specific to Keycloak
- JUnit support
- Ability to load resolved artifacts from Maven

Build setup: Building a fat jar



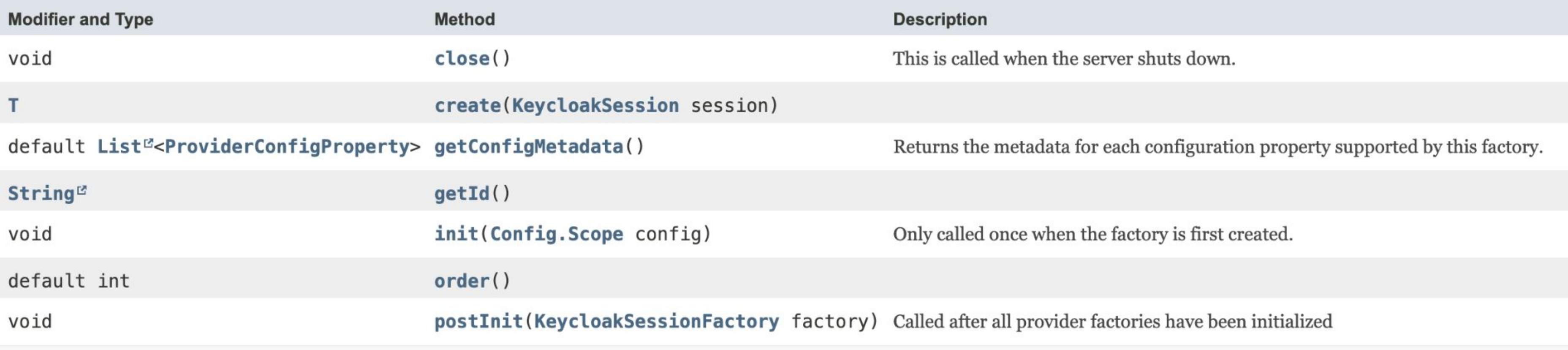
```
<plugin>
  <groupId>org.apache.maven.plugins</groupId>
  <artifactId>maven-shade-plugin</artifactId>
  <version>3.2.4</version>
  <executions>
    <execution>
      <phase>package</phase>
      <goals>
        <goal>shade</goal>
      </goals>
    </execution>
  </executions>
  <configuration>
    <filters>
      <filter>
        <artifact>*:*</artifact>
        <excludes>
          <exclude>META-INF/MANIFEST.MF</exclude>
        </excludes>
      </filter>
    </filters>
  </configuration>
</plugin>
```

To fat jar or not to fat jar?

- Depends on how you are packaging, and other if there are other extensions
- TIP: If you’re building a project with multiple extensions with overlapping dependencies, use the "old" maven EAR plugin to collect your dependencies

Lifecycle of an extension

- Every extension implements ProviderFactory and Provider
  - ProviderFactory sets up the Provider (classic factory pattern)
  - Provider does the thing!



Modifier and Type	Method	Description
`void`	`close()`	This is called when the server shuts down.
`T`	`create(KeycloakSession session)`	
`default List<ProviderConfigProperty>`	`getConfigMetadata()`	Returns the metadata for each configuration property supported by this factory.
`String`	`getId()`	
`void`	`init(Config.Scope config)`	Only called once when the factory is first created.
`default int`	`order()`	
`void`	`postInit(KeycloakSessionFactory factory)`	Called after all provider factories have been initialized

phase//

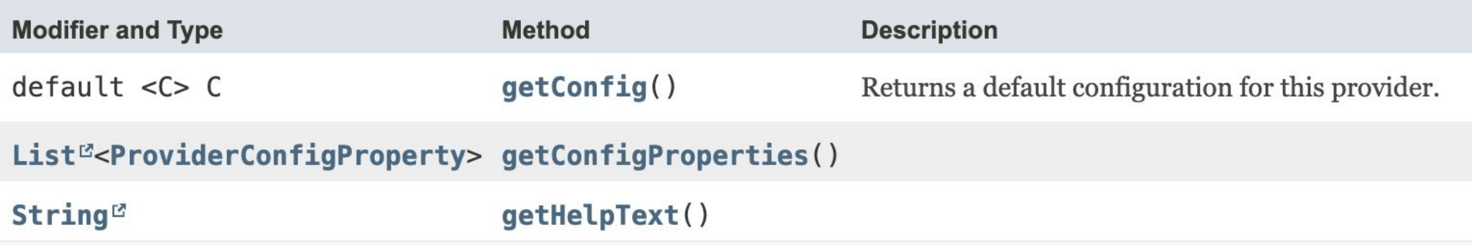
Lifecycle of an extension

Important methods:

- void init(Config.Scope config)
  - Only called once when the factory is first created
- void postInit(KeycloakSessionFactory factory)
  - Called after all provider factories have been initialized
    - E.g. Use it to create new, default authentication flows
    - E.g. Add your roles in all realms
- T create(KeycloakSession session)
  - Makes the Provider!
- void close()
  - Remember to clean up before Keycloak shuts down

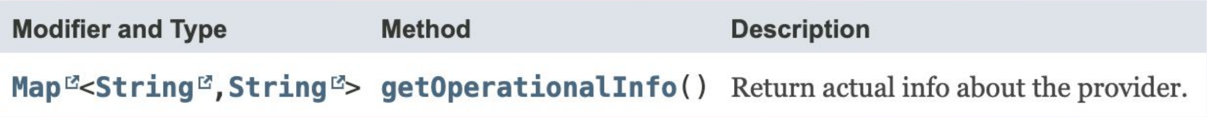
Lifecycle of an extension

- 2 additional interfaces
  - ConfiguredProvider - for many extensions that can be configured by the Keycloak UI



Modifier and Type	Method	Description
`default <C> C`	`getConfig()`	Returns a default configuration for this provider.
`List<ProviderConfigProperty>`	`getConfigProperties()`	
`String`	`getHelpText()`	

  - ServerInfoAwareProviderFactory - can show additional information about extension in Provider Info page in the Admin UI



Modifier and Type	Method	Description
`Map<String,String>`	`getOperationalInfo()`	Return actual info about the provider.

Configuration

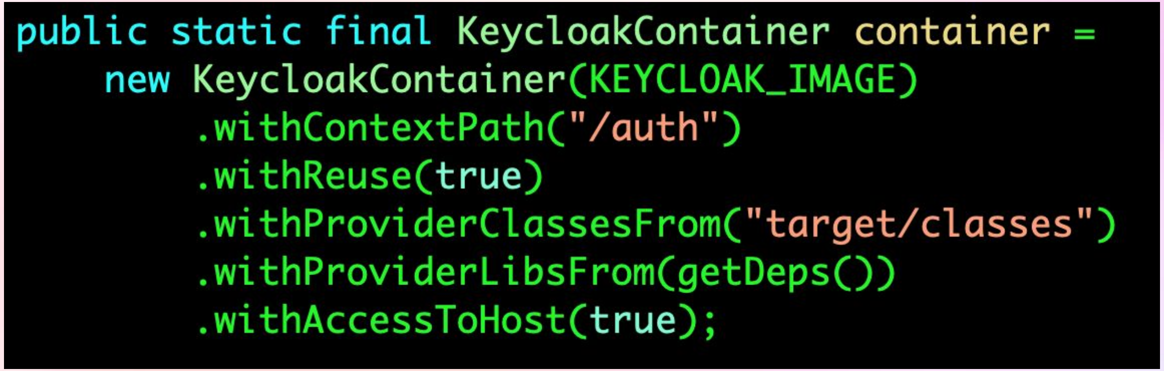
- How can I pass configuration information to my extension?
    - Keycloak’s answer: *“Let’s make it as confusing as humanly possible!”*
    - The format:
        - KC_{SPI_NAME}_{PROVIDER_ID}_{VARIABLE_NAME}
    - Example:
        - KC_SPI_AUTHENTICATOR_CUSTOM_USERNAME_PASSWORD_FORM_SELF_REGISTRATION_URL
    - Getting it out of the Config.Scope object
        - scope.get(“selfRegistrationUrl”)

Testing

- Keycloak way:
  - org.keycloak.testsuite.KeycloakServer
  - Arquillian
- Better way for extensions:
  - **Testcontainers** https://github.com/dasniko/testcontainers-keycloak
  - Cypress https://github.com/wimdeblauwe/testcontainers-cypress

Testing: Testcontainers

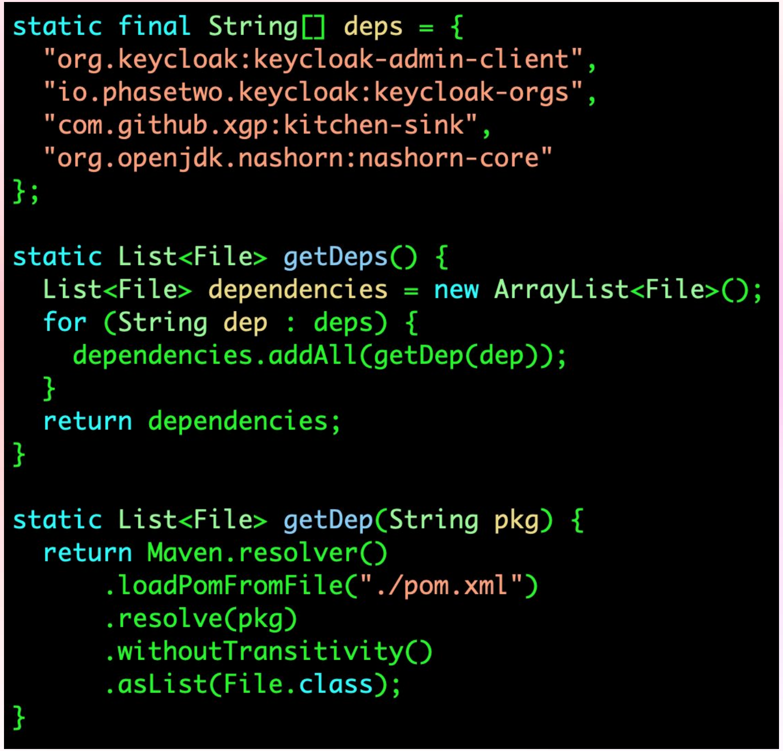
- Create a Keycloak instance:



```java
public static final KeycloakContainer container =
    new KeycloakContainer(KEYCLOAK_IMAGE)
        .withContextPath("/auth")
        .withReuse(true)
        .withProviderClassesFrom("target/classes")
        .withProviderLibsFrom(getDeps())
        .withAccessToHost(true);
```

Testing: Testcontainers

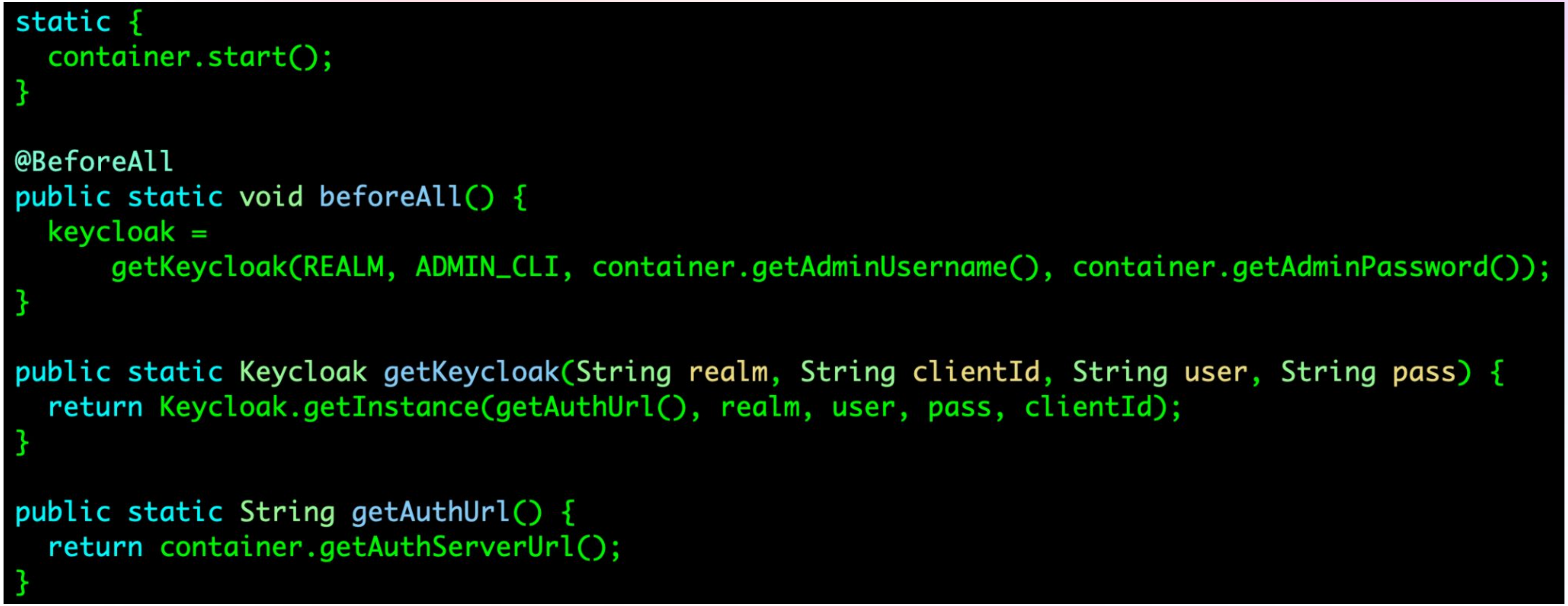
- Use dependencies defined in your pom:



```java
static final String[] deps = {
  "org.keycloak:keycloak-admin-client",
  "io.phasetwo.keycloak:keycloak-orgs",
  "com.github.xgp:kitchen-sink",
  "org.openjdk.nashorn:nashorn-core"
};

static List<File> getDeps() {
  List<File> dependencies = new ArrayList<File>();
  for (String dep : deps) {
    dependencies.addAll(getDep(dep));
  }
  return dependencies;
}

static List<File> getDep(String pkg) {
  return Maven.resolver()
      .loadPomFromFile("./pom.xml")
      .resolve(pkg)
      .withoutTransitivity()
      .asList(File.class);
}
```

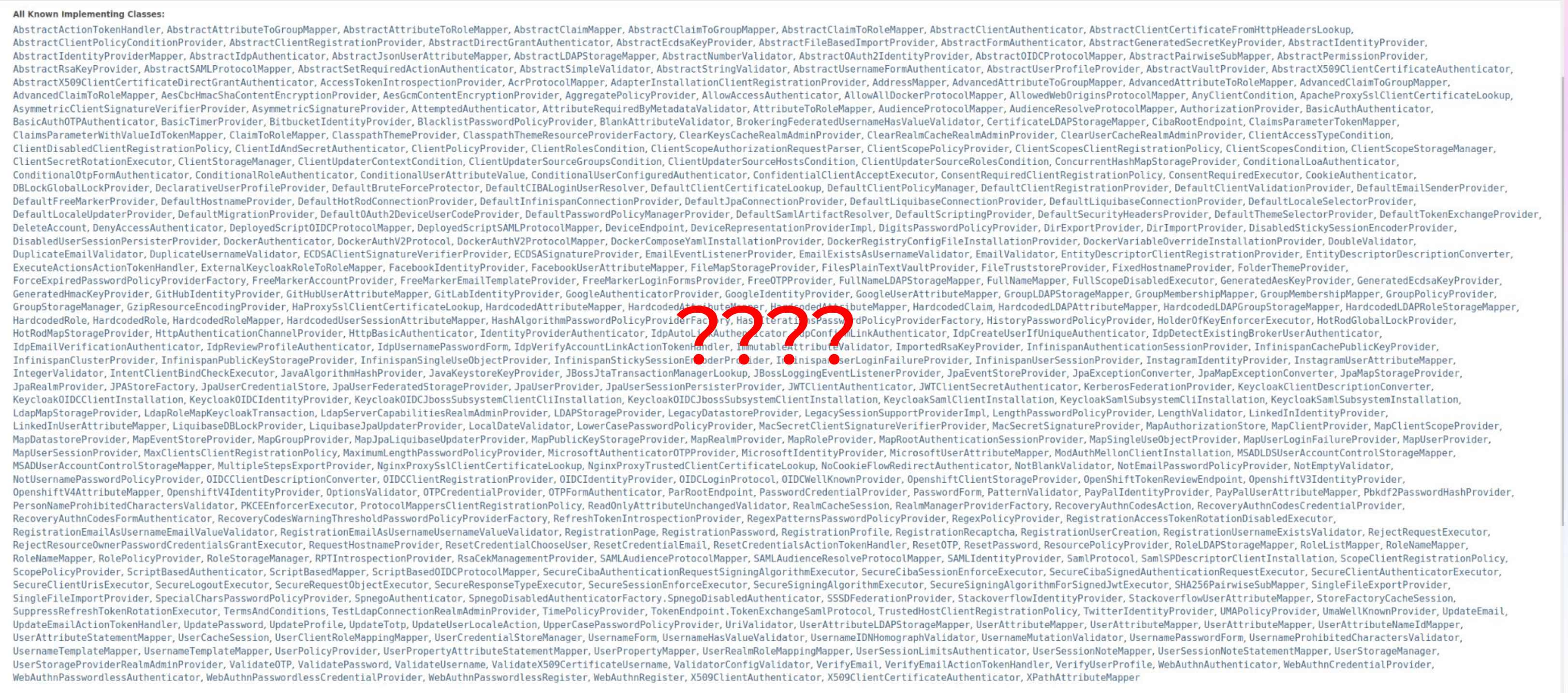
Testing: Testcontainers

- Start the container, and get a Keycloak Admin API client



```
static {
  container.start();
}

@BeforeAll
public static void beforeAll() {
  keycloak =
      getKeycloak(REALM, ADMIN_CLI, container.getAdminUsername(), container.getAdminPassword());
}

public static Keycloak getKeycloak(String realm, String clientId, String user, String pass) {
  return Keycloak.getInstance(getAuthUrl(), realm, user, pass, clientId);
}

public static String getAuthUrl() {
  return container.getAuthServerUrl();
}
```

What can I do? Where to start?



All Known Implementing Classes:

AbstractActionTokenHandler, AbstractAttributeToGroupMapper, AbstractAttributeToRoleMapper, AbstractClaimMapper, AbstractClaimToGroupMapper, AbstractClaimToRoleMapper, AbstractClientAuthenticator, AbstractClientCertificateFromHttpHeadersLookup,
AbstractClientPolicyConditionProvider, AbstractClientRegistrationProvider, AbstractDirectGrantAuthenticator, AbstractEcdsaKeyProvider, AbstractFileBasedImportProvider, AbstractFormAuthenticator, AbstractGeneratedSecretKeyProvider, AbstractIdentityProvider,
AbstractIdentityProviderMapper, AbstractIdpAuthenticator, AbstractJsonUserAttributeMapper, AbstractLDAPStorageMapper, AbstractNumberValidator, AbstractOAuth2IdentityProvider, AbstractOIDCProtocolMapper, AbstractPairwiseSubMapper, AbstractPermissionProvider,
AbstractRsaKeyProvider, AbstractSAMLProtocolMapper, AbstractSetRequiredActionAuthenticator, AbstractSimpleValidator, AbstractStringValidator, AbstractUsernameFormAuthenticator, AbstractUserProfileProvider, AbstractVaultProvider, AbstractX509ClientCertificateAuthenticator,
AbstractX509ClientCertificateDirectGrantAuthenticator, AccessTokenIntrospectionProvider, AcrProtocolMapper, AdapterInstallationClientRegistrationProvider, AddressMapper, AdvancedAttributeToGroupMapper, AdvancedAttributeToRoleMapper, AdvancedClaimToGroupMapper,
AdvancedClaimToRoleMapper, AesCbcHmacShaContentEncryptionProvider, AesGcmContentEncryptionProvider, AggregatePolicyProvider, AllowAccessAuthenticator, AllowAllDockerProtocolMapper, AllowedWebOriginsProtocolMapper, AnyClientCondition, ApacheProxySslClientCertificateLookup,
AsymmetricClientSignatureVerifierProvider, AsymmetricSignatureProvider, AttemptedAuthenticator, AttributeRequiredByMetadataValidator, AttributeToRoleMapper, AudienceProtocolMapper, AudienceResolveProtocolMapper, AuthorizationProvider, BasicAuthAuthenticator,
BasicAuthOTPAuthenticator, BasicTimerProvider, BitbucketIdentityProvider, BlacklistPasswordPolicyProvider, BlankAttributeValidator, BrokeringFederatedUsernameHasValueValidator, CertificateLDAPStorageMapper, CibaRootEndpoint, ClaimsParameterTokenMapper,
ClaimsParameterWithValueIdTokenMapper, ClaimToRoleMapper, ClasspathThemeProvider, ClasspathThemeResourceProviderFactory, ClearKeysCacheRealmAdminProvider, ClearRealmCacheRealmAdminProvider, ClearUserCacheRealmAdminProvider, ClientAccessTypeCondition,
ClientDisabledClientRegistrationPolicy, ClientIdAndSecretAuthenticator, ClientPolicyProvider, ClientRolesCondition, ClientScopeAuthorizationRequestParser, ClientScopePolicyProvider, ClientScopesClientRegistrationPolicy, ClientScopesCondition, ClientScopeStorageManager,
ClientSecretRotationExecutor, ClientStorageManager, ClientUpdaterContextCondition, ClientUpdaterSourceGroupsCondition, ClientUpdaterSourceHostsCondition, ClientUpdaterSourceRolesCondition, ConcurrentHashMapStorageProvider, ConditionalLoaAuthenticator,
ConditionalOtpFormAuthenticator, ConditionalRoleAuthenticator, ConditionalUserAttributeValue, ConditionalUserConfiguredAuthenticator, ConfidentialClientAcceptExecutor, ConsentRequiredClientRegistrationPolicy, ConsentRequiredExecutor, CookieAuthenticator,
DBLockGlobalLockProvider, DeclarativeUserProfileProvider, DefaultBruteForceProtector, DefaultCIBALoginUserResolver, DefaultClientCertificateLookup, DefaultClientPolicyManager, DefaultClientRegistrationProvider, DefaultClientValidationProvider, DefaultEmailSenderProvider,
DefaultFreeMarkerProvider, DefaultHostnameProvider, DefaultHotRodConnectionProvider, DefaultInfinispanConnectionProvider, DefaultJpaConnectionProvider, DefaultLiquibaseConnectionProvider, DefaultLiquibaseConnectionProvider, DefaultLocaleSelectorProvider,
DefaultLocaleUpdaterProvider, DefaultMigrationProvider, DefaultOAuth2DeviceUserCodeProvider, DefaultPasswordPolicyManagerProvider, DefaultSamlArtifactResolver, DefaultScriptingProvider, DefaultSecurityHeadersProvider, DefaultThemeSelectorProvider, DefaultTokenExchangeProvider,
DeleteAccount, DenyAccessAuthenticator, DeployedScriptOIDCProtocolMapper, DeployedScriptSAMLProtocolMapper, DeviceEndpoint, DeviceRepresentationProviderImpl, DigitsPasswordPolicyProvider, DirExportProvider, DirImportProvider, DisabledStickySessionEncoderProvider,
DisabledUserSessionPersisterProvider, DockerAuthenticator, DockerAuthV2Protocol, DockerAuthV2ProtocolMapper, DockerComposeYamlInstallationProvider, DockerRegistryConfigFileInstallationProvider, DockerVariableOverrideInstallationProvider, DoubleValidator,
DuplicateEmailValidator, DuplicateUsernameValidator, ECDSAClientSignatureVerifierProvider, ECDSASignatureProvider, EmailEventListenerProvider, EmailExistsAsUsernameValidator, EmailValidator, EntityDescriptorClientRegistrationProvider, EntityDescriptorDescriptionConverter,
ExecuteActionsActionTokenHandler, ExternalKeycloakRoleToRoleMapper, FacebookIdentityProvider, FacebookUserAttributeMapper, FileMapStorageProvider, FilesPlainTextVaultProvider, FileTruststoreProvider, FixedHostnameProvider, FolderThemeProvider,
ForceExpiredPasswordPolicyProviderFactory, FreeMarkerAccountProvider, FreeMarkerEmailTemplateProvider, FreeMarkerLoginFormsProvider, FreeOTPProvider, FullNameLDAPStorageMapper, FullNameMapper, FullScopeDisabledExecutor, GeneratedAesKeyProvider, GeneratedEcdsaKeyProvider,
GeneratedHmacKeyProvider, GitHubIdentityProvider, GitHubUserAttributeMapper, GitLabIdentityProvider, GoogleAuthenticatorProvider, GoogleIdentityProvider, GoogleUserAttributeMapper, GroupLDAPStorageMapper, GroupMembershipMapper, GroupMembershipMapper, GroupPolicyProvider,
GroupStorageManager, GzipResourceEncodingProvider, HaProxySslClientCertificateLookup, HardcodedAttributeMapper, HardcodedAttributeMapper, HardcodedAttributeMapper, HardcodedClaim, HardcodedLDAPAttributeMapper, HardcodedLDAPGroupStorageMapper, HardcodedLDAPRoleStorageMapper,
HardcodedRole, HardcodedRole, HardcodedRoleMapper, HardcodedUserSessionAttributeMapper, HashAlgorithmPasswordPolicyProviderFactory, HashIterationsPasswordPolicyProviderFactory, HistoryPasswordPolicyProvider, HolderOfKeyEnforcerExecutor, HotRodGlobalLockProvider,
HotRodMapStorageProvider, HttpAuthenticationChannelProvider, HttpBasicAuthenticator, IdentityProviderAuthenticator, IdpAutoLinkAuthenticator, IdpConfirmLinkAuthenticator, IdpCreateUserIfUniqueAuthenticator, IdpDetectExistingBrokerUserAuthenticator,
IdpEmailVerificationAuthenticator, IdpReviewProfileAuthenticator, IdpUsernamePasswordForm, IdpVerifyAccountLinkActionTokenHandler, ImmutableAttributeValidator, ImportedRsaKeyProvider, InfinispanAuthenticationSessionProvider, InfinispanCachePublicKeyProvider,
InfinispanClusterProvider, InfinispanPublicKeyStorageProvider, InfinispanSingleUseObjectProvider, InfinispanStickySessionEncoderProvider, InfinispanUserLoginFailureProvider, InfinispanUserSessionProvider, InstagramIdentityProvider, InstagramUserAttributeMapper,
IntegerValidator, IntentClientBindCheckExecutor, JavaAlgorithmHashProvider, JavaKeystoreKeyProvider, JBossJtaTransactionManagerLookup, JBossLoggingEventListenerProvider, JpaEventStoreProvider, JpaExceptionConverter, JpaMapExceptionConverter, JpaMapStorageProvider,
JpaRealmProvider, JPAStoreFactory, JpaUserCredentialStore, JpaUserFederatedStorageProvider, JpaUserProvider, JpaUserSessionPersisterProvider, JWTClientAuthenticator, JWTClientSecretAuthenticator, KerberosFederationProvider, KeycloakClientDescriptionConverter,
KeycloakOIDCClientInstallation, KeycloakOIDCIdentityProvider, KeycloakOIDCJbossSubsystemClientCliInstallation, KeycloakOIDCJbossSubsystemClientInstallation, KeycloakSamlClientInstallation, KeycloakSamlSubsystemCliInstallation, KeycloakSamlSubsystemInstallation,
LdapMapStorageProvider, LdapRoleMapKeycloakTransaction, LdapServerCapabilitiesRealmAdminProvider, LDAPStorageProvider, LegacyDatastoreProvider, LegacySessionSupportProviderImpl, LengthPasswordPolicyProvider, LengthValidator, LinkedInIdentityProvider,
LinkedInUserAttributeMapper, LiquibaseDBLockProvider, LiquibaseJpaUpdaterProvider, LocalDateValidator, LowerCasePasswordPolicyProvider, MacSecretClientSignatureVerifierProvider, MacSecretSignatureProvider, MapAuthorizationStore, MapClientProvider, MapClientScopeProvider,
MapDatastoreProvider, MapEventStoreProvider, MapGroupProvider, MapJpaLiquibaseUpdaterProvider, MapPublicKeyStorageProvider, MapRealmProvider, MapRoleProvider, MapRootAuthenticationSessionProvider, MapSingleUseObjectProvider, MapUserLoginFailureProvider, MapUserProvider,
MapUserSessionProvider, MaxClientsClientRegistrationPolicy, MaximumLengthPasswordPolicyProvider, MicrosoftAuthenticatorOTPProvider, MicrosoftIdentityProvider, MicrosoftUserAttributeMapper, ModAuthMellonClientInstallation, MSADLDSUserAccountControlStorageMapper,
MSADUserAccountControlStorageMapper, MultipleStepsExportProvider, NginxProxySslClientCertificateLookup, NginxProxyTrustedClientCertificateLookup, NoCookieFlowRedirectAuthenticator, NotBlankValidator, NotEmailPasswordPolicyProvider, NotEmptyValidator,
NotUsernamePasswordPolicyProvider, OIDCClientDescriptionConverter, OIDCClientRegistrationProvider, OIDCIdentityProvider, OIDCLoginProtocol, OIDCWellKnownProvider, OpenshiftClientStorageProvider, OpenShiftTokenReviewEndpoint, OpenshiftV3IdentityProvider,
OpenshiftV4AttributeMapper, OpenshiftV4IdentityProvider, OptionsValidator, OTPCredentialProvider, OTPFormAuthenticator, ParRootEndpoint, PasswordCredentialProvider, PasswordForm, PatternValidator, PayPalIdentityProvider, PayPalUserAttributeMapper, Pbkdf2PasswordHashProvider,
PersonNameProhibitedCharactersValidator, PKCEEnforcerExecutor, ProtocolMappersClientRegistrationPolicy, ReadOnlyAttributeUnchangedValidator, RealmCacheSession, RealmManagerProviderFactory, RecoveryAuthnCodesAction, RecoveryAuthnCodesCredentialProvider,
RecoveryAuthnCodesFormAuthenticator, RecoveryCodesWarningThresholdPasswordPolicyProviderFactory, RefreshTokenIntrospectionProvider, RegexPatternsPasswordPolicyProvider, RegexPolicyProvider, RegistrationAccessTokenRotationDisabledExecutor,
RegistrationEmailAsUsernameEmailValueValidator, RegistrationEmailAsUsernameUsernameValueValidator, RegistrationPage, RegistrationPassword, RegistrationProfile, RegistrationRecaptcha, RegistrationUserCreation, RegistrationUsernameExistsValidator, RejectRequestExecutor,
RejectResourceOwnerPasswordCredentialsGrantExecutor, RequestHostnameProvider, ResetCredentialChooseUser, ResetCredentialEmail, ResetCredentialsActionTokenHandler, ResetOTP, ResetPassword, ResourcePolicyProvider, RoleLDAPStorageMapper, RoleListMapper, RoleNameMapper,
RoleNameMapper, RolePolicyProvider, RoleStorageManager, RPTIntrospectionProvider, RsaCekManagementProvider, SAMLAudienceProtocolMapper, SAMLAudienceResolveProtocolMapper, SAMLIdentityProvider, SamlProtocol, SamlSPDescriptorClientInstallation, ScopeClientRegistrationPolicy,
ScopePolicyProvider, ScriptBasedAuthenticator, ScriptBasedMapper, ScriptBasedOIDCProtocolMapper, SecureCibaAuthenticationRequestSigningAlgorithmExecutor, SecureCibaSessionEnforceExecutor, SecureCibaSignedAuthenticationRequestExecutor, SecureClientAuthenticatorExecutor,
SecureClientUrisExecutor, SecureLogoutExecutor, SecureRequestObjectExecutor, SecureResponseTypeExecutor, SecureSessionEnforceExecutor, SecureSigningAlgorithmExecutor, SecureSigningAlgorithmForSignedJwtExecutor, SHA256PairwiseSubMapper, SingleFileExportProvider,
SingleFileImportProvider, SpecialCharsPasswordPolicyProvider, SpnegoAuthenticator, SpnegoDisabledAuthenticatorFactory.SpnegoDisabledAuthenticator, SSSDFederationProvider, StackoverflowIdentityProvider, StackoverflowUserAttributeMapper, StoreFactoryCacheSession,
SuppressRefreshTokenRotationExecutor, TermsAndConditions, TestLdapConnectionRealmAdminProvider, TimePolicyProvider, TokenEndpoint.TokenExchangeSamlProtocol, TrustedHostClientRegistrationPolicy, TwitterIdentityProvider, UMAPolicyProvider, UmaWellKnownProvider, UpdateEmail,
UpdateEmailActionTokenHandler, UpdatePassword, UpdateProfile, UpdateTotp, UpdateUserLocaleAction, UpperCasePasswordPolicyProvider, UriValidator, UserAttributeLDAPStorageMapper, UserAttributeMapper, UserAttributeMapper, UserAttributeMapper, UserAttributeNameIdMapper,
UserAttributeStatementMapper, UserCacheSession, UserClientRoleMappingMapper, UserCredentialStoreManager, UsernameForm, UsernameHasValueValidator, UsernameIDNHomographValidator, UsernameMutationValidator, UsernamePasswordForm, UsernameProhibitedCharactersValidator,
UsernameTemplateMapper, UsernameTemplateMapper, UserPolicyProvider, UserPropertyAttributeStatementMapper, UserPropertyMapper, UserRealmRoleMappingMapper, UserSessionLimitsAuthenticator, UserSessionNoteMapper, UserSessionNoteStatementMapper, UserStorageManager,
UserStorageProviderRealmAdminProvider, ValidateOTP, ValidatePassword, ValidateUsername, ValidateX509CertificateUsername, ValidatorConfigValidator, VerifyEmail, VerifyEmailActionTokenHandler, VerifyUserProfile, WebAuthnAuthenticator, WebAuthnCredentialProvider,
WebAuthnPasswordlessAuthenticator, WebAuthnPasswordlessCredentialProvider, WebAuthnPasswordlessRegister, WebAuthnRegister, X509ClientAuthenticator, X509ClientCertificateAuthenticator, XPathAttributeMapper

A real example

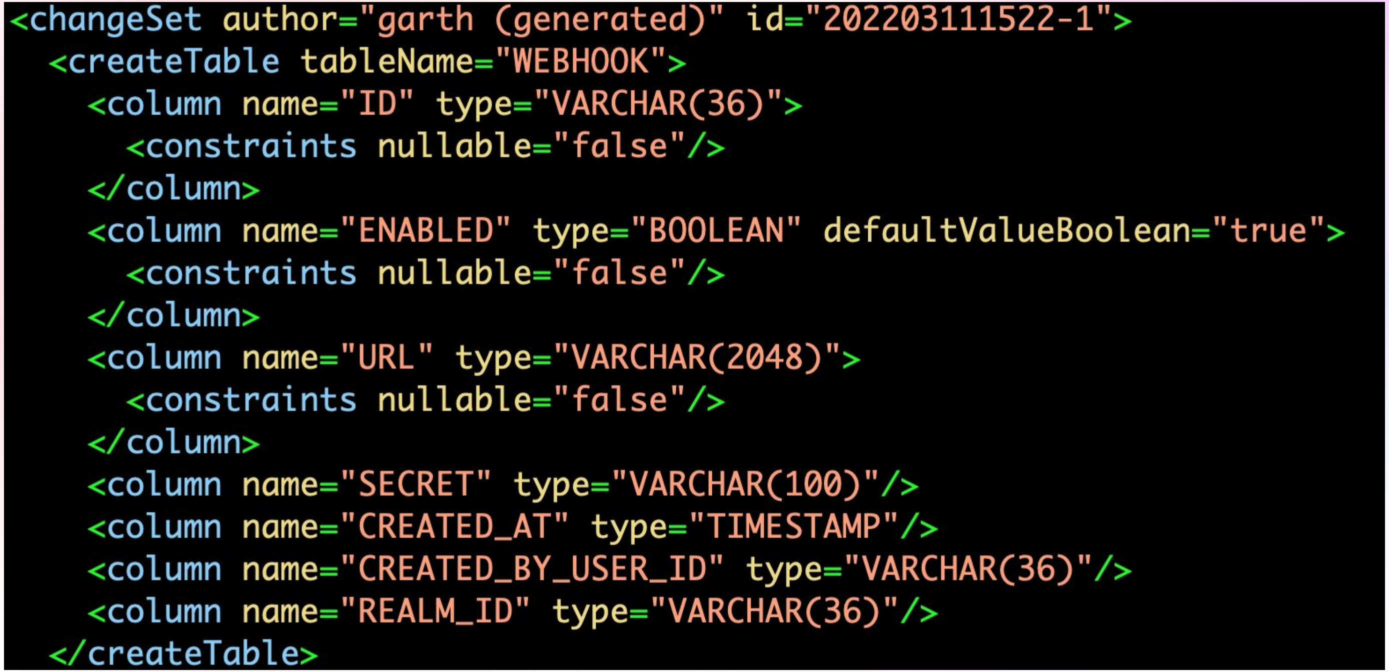
- We're going to build a generally useful Webhooks extension to the Keycloak event system (like Stripe and most modern APIs)
- In the process, we will use Keycloak extension SPIs to implement features:
    - **JPA entities** so Webhook definitions can be persisted
    - **Custom REST resources** to create an API for managing Webhook subscriptions
    - **Event listener** to capture Keycloak events and dispatch them to Webhook endpoints
    - **[BONUS] Admin UI** to create, manage and view the Webhooks
- (I'm going to go a bit fast, as all of the code is in our open source extensions)

JPA entities

- Store entities in the database like Keycloak
  - Create a liquibase migration script
  - Create your JPA entities
  - Create the JpaEntityProvider implementation and register your entities and migration script
  - Create Model classes to wrap the entities
- [Bonus] Creating our own SPI so that we can provide a convenient/protected way of accessing Model classes

JPA entities

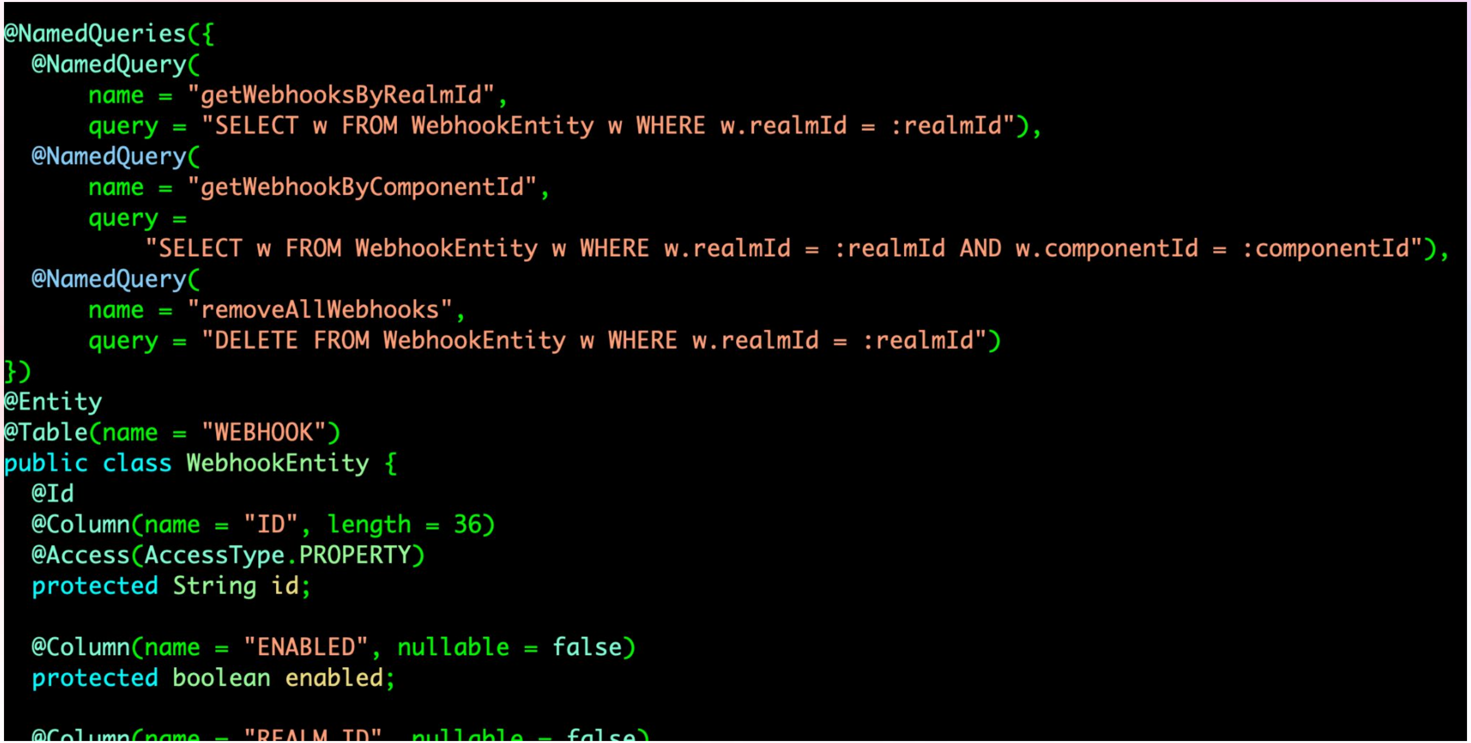
- Create a liquibase migration script
  - ○ Put it in src/main/resources/META-INF so it gets packaged



```xml
<changeSet author="garth (generated)" id="202203111522-1">
  <createTable tableName="WEBHOOK">
    <column name="ID" type="VARCHAR(36)">
      <constraints nullable="false"/>
    </column>
    <column name="ENABLED" type="BOOLEAN" defaultValueBoolean="true">
      <constraints nullable="false"/>
    </column>
    <column name="URL" type="VARCHAR(2048)">
      <constraints nullable="false"/>
    </column>
    <column name="SECRET" type="VARCHAR(100)"/>
    <column name="CREATED_AT" type="TIMESTAMP"/>
    <column name="CREATED_BY_USER_ID" type="VARCHAR(36)"/>
    <column name="REALM_ID" type="VARCHAR(36)"/>
  </createTable>
```

JPA entities

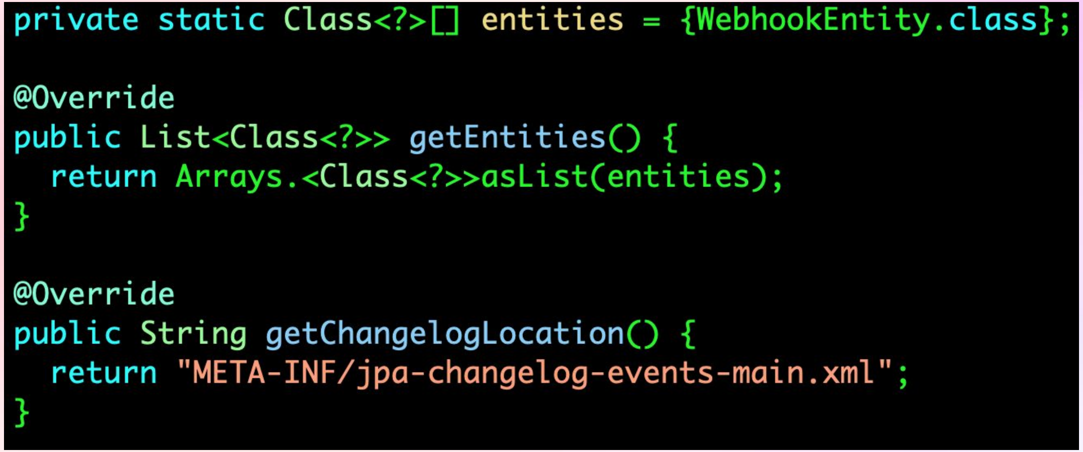
- Create your JPA entities



```
@NamedQueries({
  @NamedQuery(
      name = "getWebhooksByRealmId",
      query = "SELECT w FROM WebhookEntity w WHERE w.realmId = :realmId"),
  @NamedQuery(
      name = "getWebhookByComponentId",
      query =
          "SELECT w FROM WebhookEntity w WHERE w.realmId = :realmId AND w.componentId = :componentId"),
  @NamedQuery(
      name = "removeAllWebhooks",
      query = "DELETE FROM WebhookEntity w WHERE w.realmId = :realmId")
})
@Entity
@Table(name = "WEBHOOK")
public class WebhookEntity {
  @Id
  @Column(name = "ID", length = 36)
  @Access(AccessType.PROPERTY)
  protected String id;

  @Column(name = "ENABLED", nullable = false)
  protected boolean enabled;

  @Column(name = "REALM_ID", nullable = false)
```

JPA entities

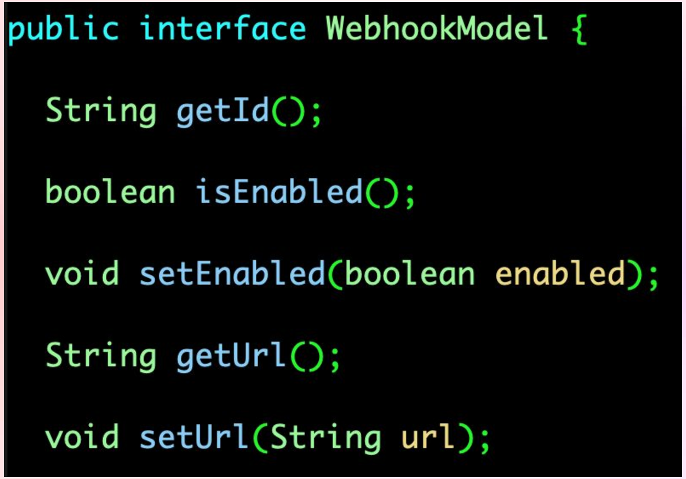
- Create the JpaEntityProvider implementation and register your entities and migration script



```java
private static Class<?>[] entities = {WebhookEntity.class};

@Override
public List<Class<?>> getEntities() {
  return Arrays.<Class<?>>asList(entities);
}

@Override
public String getChangelogLocation() {
  return "META-INF/jpa-changelog-events-main.xml";
}
```

JPA entities

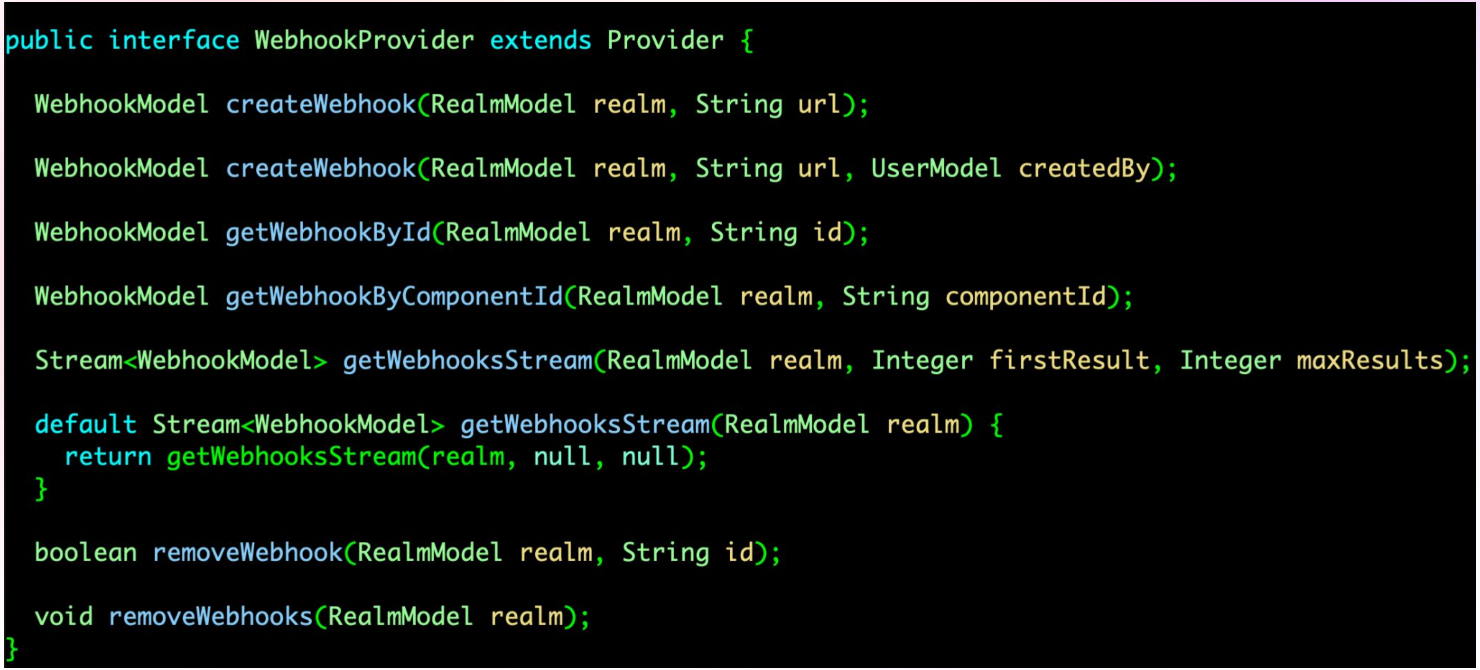
- Create Model interfaces and implementations to wrap the entities



```
public interface WebhookModel {

  String getId();

  boolean isEnabled();

  void setEnabled(boolean enabled);

  String getUrl();

  void setUrl(String url);
```

JPA entities

- Bonus: Creating our own SPI so that we can provide a convenient/protected way of accessing Model classes



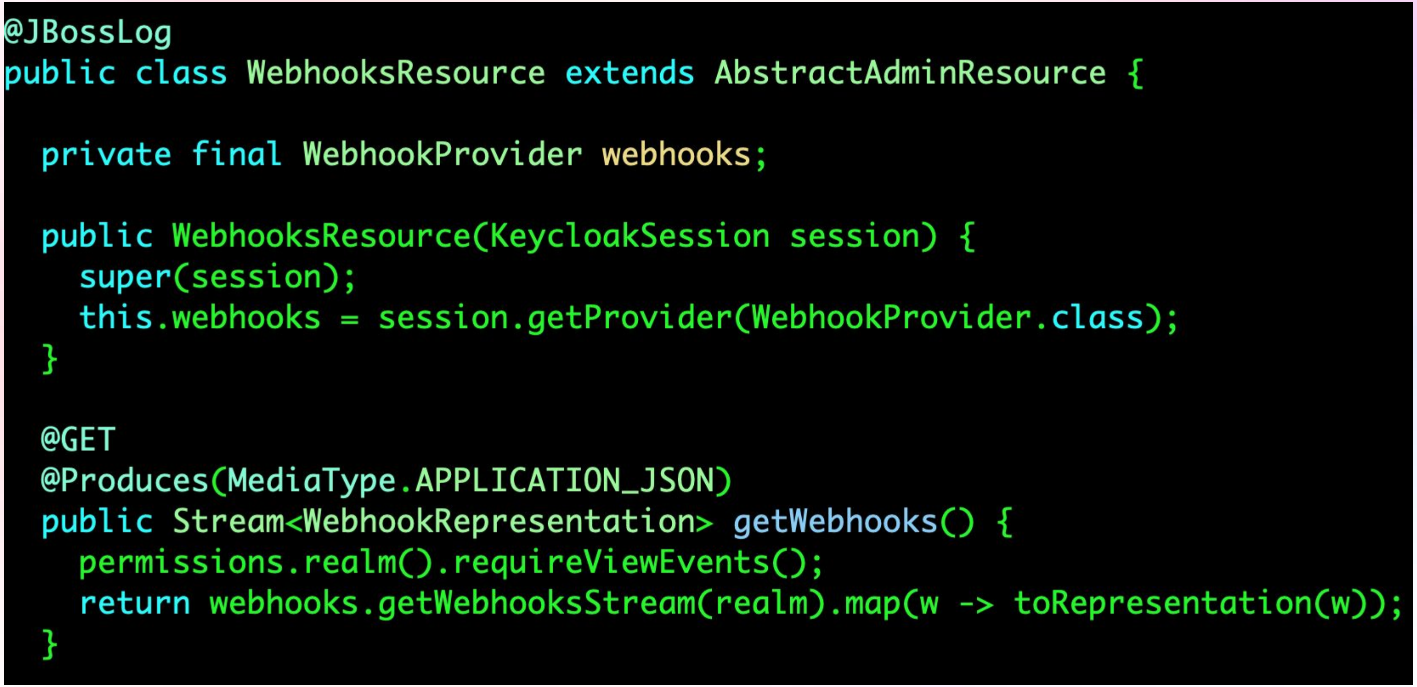
```
public interface WebhookProvider extends Provider {

  WebhookModel createWebhook(RealmModel realm, String url);

  WebhookModel createWebhook(RealmModel realm, String url, UserModel createdBy);

  WebhookModel getWebhookById(RealmModel realm, String id);

  WebhookModel getWebhookByComponentId(RealmModel realm, String componentId);

  Stream<WebhookModel> getWebhooksStream(RealmModel realm, Integer firstResult, Integer maxResults);

  default Stream<WebhookModel> getWebhooksStream(RealmModel realm) {
    return getWebhooksStream(realm, null, null);
  }

  boolean removeWebhook(RealmModel realm, String id);

  void removeWebhooks(RealmModel realm);
}
```

Custom REST resources

- Implement RealmResourceProvider to provide an API for Webhook subscriptions
  - The getResource() method returns a standard JAX-RS implementation
  - We need to remember to handle access control (Keycloak doesn’t do it for us)
  - Make sure we’re giving auditability of our changes by adding Admin events

Custom REST resources

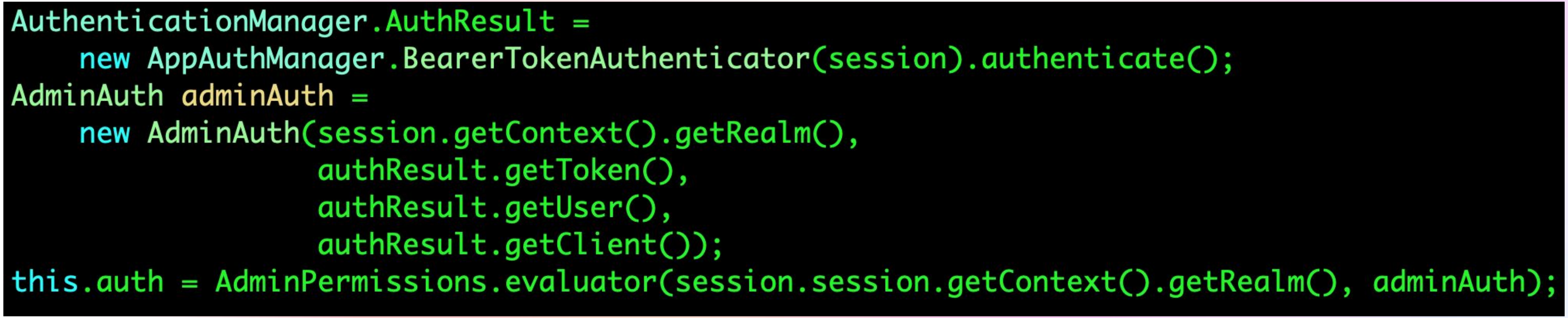
- The getResource() method returns a standard JAX-RS implementation



```java
@JBossLog
public class WebhooksResource extends AbstractAdminResource {

  private final WebhookProvider webhooks;

  public WebhooksResource(KeycloakSession session) {
    super(session);
    this.webhooks = session.getProvider(WebhookProvider.class);
  }

  @GET
  @Produces(MediaType.APPLICATION_JSON)
  public Stream<WebhookRepresentation> getWebhooks() {
    permissions.realm().requireViewEvents();
    return webhooks.getWebhooksStream(realm).map(w -> toRepresentation(w));
  }
```

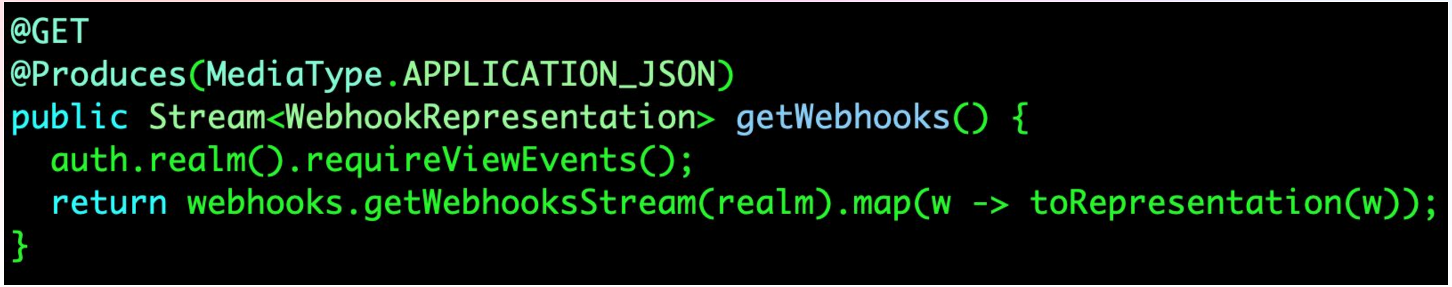
Custom REST resources

- We need to remember to handle access control (Keycloak doesn’t do it for us)



```
AuthenticationManager.AuthResult =
    new AppAuthManager.BearerTokenAuthenticator(session).authenticate();
AdminAuth adminAuth =
    new AdminAuth(session.getContext().getRealm(),
                  authResult.getToken(),
                  authResult.getUser(),
                  authResult.getClient());
this.auth = AdminPermissions.evaluator(session.session.getContext().getRealm(), adminAuth);
```

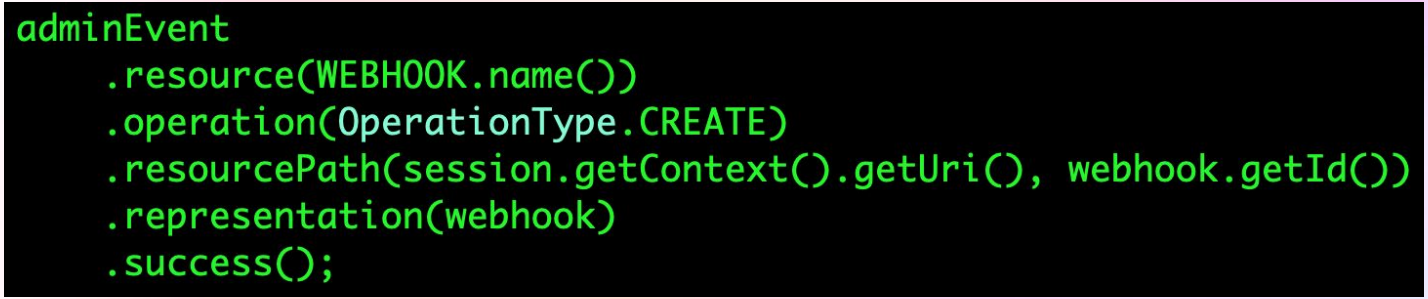
- Then we can do easy access control checks like:



```
@GET
@Produces(MediaType.APPLICATION_JSON)
public Stream<WebhookRepresentation> getWebhooks() {
  auth.realm().requireViewEvents();
  return webhooks.getWebhooksStream(realm).map(w -> toRepresentation(w));
}
```

Custom REST resources

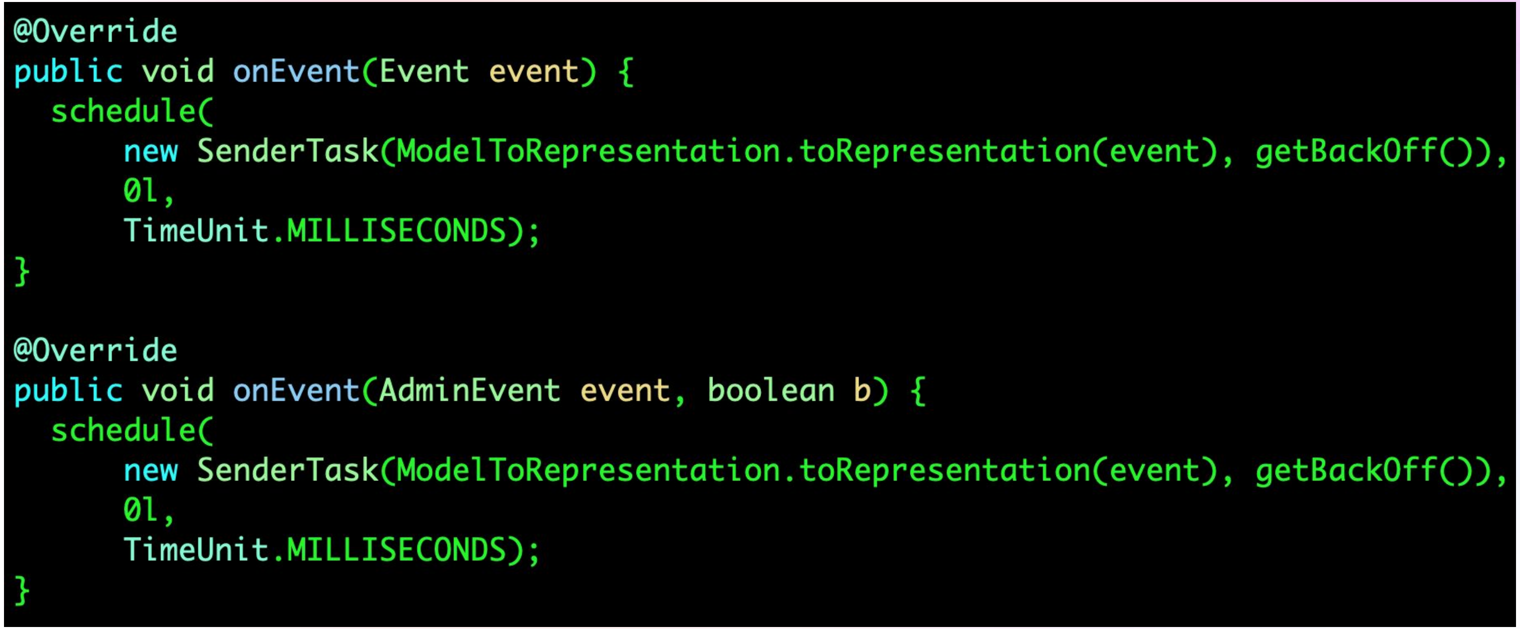
- Make sure we're giving auditability of our changes by adding Admin events



```
adminEvent
     .resource(WEBHOOK.name())
     .operation(OperationType.CREATE)
     .resourcePath(session.getContext().getUri(), webhook.getId())
     .representation(webhook)
     .success();
```

Event listener

- Capture Keycloak events and dispatch them to Webhook endpoints
  - Implement an EventListenerProvider



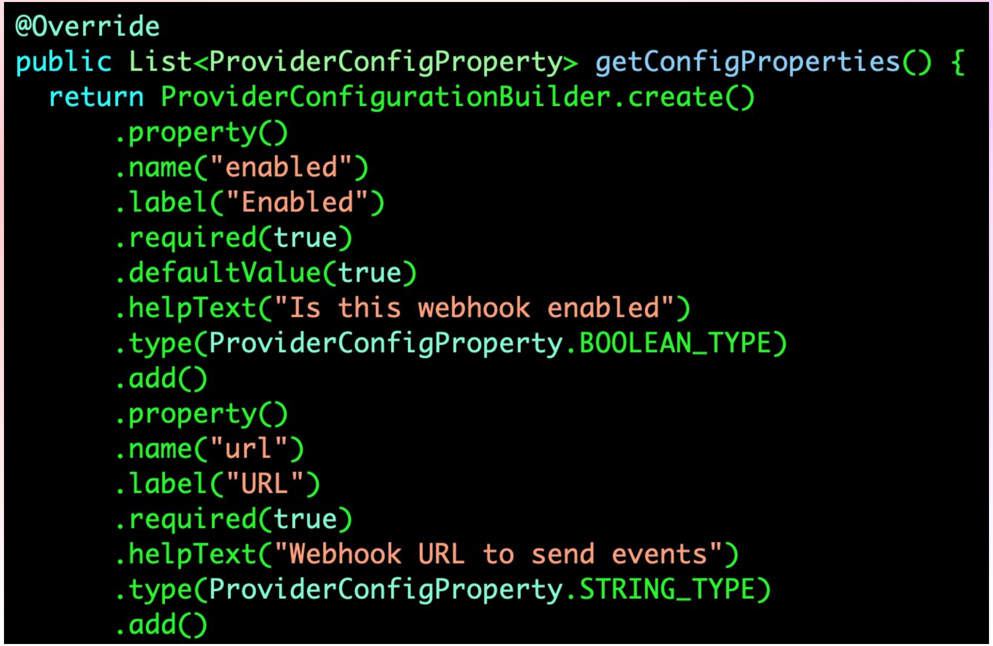
```
@Override
public void onEvent(Event event) {
  schedule(
      new SenderTask(ModelToRepresentation.toRepresentation(event), getBackOff()),
      0l,
      TimeUnit.MILLISECONDS);
}

@Override
public void onEvent(AdminEvent event, boolean b) {
  schedule(
      new SenderTask(ModelToRepresentation.toRepresentation(event), getBackOff()),
      0l,
      TimeUnit.MILLISECONDS);
}
```

Bonus: Admin UI

- Create, manage and view the Webhooks
- Using the “new” Java-based Admin UI extension mechanism
    - Note that this is essentially a copy of of how user storage providers were configured, but “generalized”
- This requires us to use/implement UiPageProviderFactory / ComponentFactory
    - There will be duplicative WebhookModels and ComponentModels, as this extension relies on components for storage
        - This is done via the preRemove, onCreate and onUpdate methods
        - And we have to go back into our REST resources and update the ComponentModels
        - And for people who were using this before, we have to migrate
        - And.... (sigh)

Bonus: Admin UI

- This uses the ConfiguredProvider interface mentioned earlier that allows us to specify properties and their types, attributes and other metadata, so that a UI can be automatically generated.



```
@Override
public List<ProviderConfigProperty> getConfigProperties() {
  return ProviderConfigurationBuilder.create()
      .property()
      .name("enabled")
      .label("Enabled")
      .required(true)
      .defaultValue(true)
      .helpText("Is this webhook enabled")
      .type(ProviderConfigProperty.BOOLEAN_TYPE)
      .add()
      .property()
      .name("url")
      .label("URL")
      .required(true)
      .helpText("Webhook URL to send events")
      .type(ProviderConfigProperty.STRING_TYPE)
      .add()
```

Demo

- Show Provider Info
- Enable event listener
- Create a Webhook
- Trigger an event
- Success!

Special thanks…

- **Our community contributors**
- The Keycloak **maintainers**, **authors** and **contributors**
- @dteleguin for **beercloak**
- @thomasdarimont for **Awesome Keycloak** and so many excellent **examples**
- @sventorben for his extensions and another talk that inspired this
- @dasniko for patience, examples, general awesomeness
- @adorsys for this **great event**
- **The whole Keycloak community**!

Questions?

https://rb.gy/tfde4g

More resources:

- Homepage: https://phasetwo.io
- GitHub: https://github.com/p2-inc
- Webhooks / Events extension: https://github.com/p2-inc/keycloak-events