  - Change the default Elasticsearch module to Solr as described after the upgrade is finished.
  - Reindex.

## 6.5 Configuration Admin

In earlier versions of Liferay, global configurations occured through **portal-ext.properties**. Problem: The Problem? The only way to know what was available to change was to refer to the giant **portal.properties**.

In Liferay 7, there is a better way to manage the configurations through the **Configuration Admin** services provided by OSGi.

### 6.5.1 Make your applications configurable - Basic Concepts

- *Typed* configuration
  - Integer
  - List of strings
  - Any other native Java types
  - Custom types
- Concept of components
  - `@Component` annotation
  - Configuration class
  - Other properties
- Scoped configuration
  - System
  - Virtual Instance
  - Site
  - Portlet Instance

### 6.5.2 Creating a System scope configuration

In the following example, we will show the minimum amount of code you need to write to make your application configurable the Liferay 7 way.

### Exercise: Creating a System scope configuration

1. **Create** a new interface named `com.example.api.ColorConfiguration` in the previously created portlet project in Eclipse.
2. **Paste** the snippet located in `snippet1.txt` in the newly created interface.

There are two Java annotations to provide metadata about the configuration.

- `@Meta.OCD` Meta.OCD (OCD stands for object class definition): Registers this class as a configuration with a specific id. You can choose any string you want, but make sure it's unique. A common pattern is to use the fully-qualified class name.
- `@Meta.AD` Meta.AD (AD stands for attribute definition): Allows specifying the default value of a configuration field as well as whether it is required or not. Note that if you set a field as "required" and don't specify a default value, it has to be done later by the system administrator for your application to work properly.

### Exercise: Import the dependencies

In order to use these annotations in your modules, you will need to create a dependency with the bnd library.

1. **Open** the `build.gradle` file and **paste** the content **snippet2.txt** at the end of the dependency section. `compile group: "biz.aQute.bnd", name: "biz.aQute.bndlib", version: "3.1.0"`
2. **Add** the following line at the end of the `bnd.bnd` file `-metatype: *`. In Eclispe, **right-click** on your project → Gradle → Refresh Gradle project to update your project's dependencies.

## 6.5.3 Your interface class

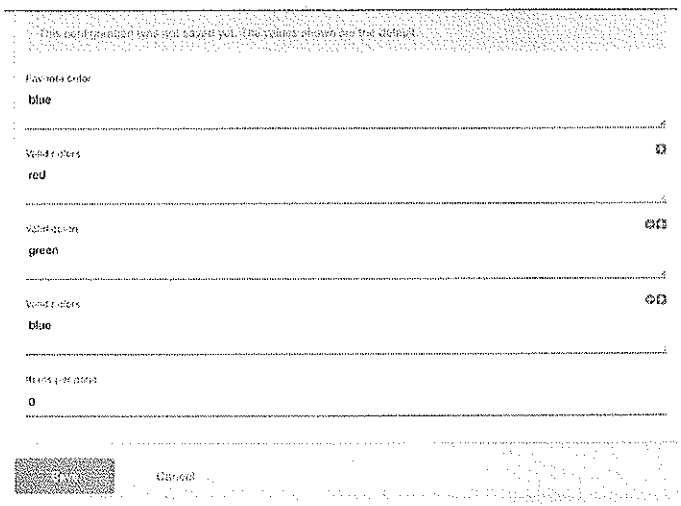
```
@Meta.OCD(id = "com.example.api.ColorConfiguration")
public interface ColorConfiguration {
    @Meta.AD(
            deflt = "blue",
            required = false
```

```
        )
    public String favoriteColor();

    @Meta.AD(
        deflt = "red|green|blue",
        required = false
        )
    public String[] validLanguages();

    @Meta.AD(required = false)
    public int itemsPerPage();
}
```

## Exercise: Configuration in the control panel

Now **navigate** to the Control Panel and then **click** on *System* → *System Settings*. Then **click** on *Other*, **find** the **ColorConfiguration** link, and **click** on it. You will be able to see the properties of ColorConfiguration as follows:
*(Figure 6.15, page 217)*

Figure 6.15:



### 6.5.4 How Can I Read the Configuration from my Application's Code?

In a component class, it uses the configuration with id `com.example.api.ColorConfigurat`
As a result, the method activate will be invoked with the application starts (due to the `@Activate` annotation) and whenever the configuration is modified (due to the `@Modified` annotation).

The activate method then uses the method `ConfigurableUtil.createConfigurable()` to convert a map of properties with the configuration to our typed class, which is easier to handle.

#### Example

Here is a simple example to illustrate how to read the configuration with a component.

```
@Component(configurationPid = "com.example.api.ColorConfiguration")
public class MyAppManager {

    public String getFavoriteColor(Map colors) {
       return colors.get(_configuration.favoriteColor());
    }

    @Activate
    @Modified
    protected void activate(Map<String, Object> properties) {
        _configuration = Configurable.createConfigurable(
        ColorConfiguration.class, properties);
    }

    private volatile ColorConfiguration _configuration;

}
```

#### Exercise: Accessing your configuration in a JSP portlet application

In Liferay, it's very common to read a configuration from a portlet class. If the portlet is a JSP portlet, the configuration object can be added to the

request object so that configurations can be read from the JSPs that comprise the application's View layer.

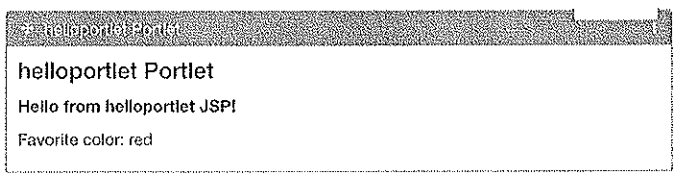
1. Go back to your portlet class (example here helloportletmvcportlet-Portlet).
2. Open snippet3.txt, copy the code in section 1, and paste it inside the component annotation.
3. Copy the code in section 2 and paste it at the end of your portlet class.
4. Copy the code in section 3 and paste it inside the doView method right before `super.doView(renderRequest, renderResponse);`.

### Exercise: Read the configuration

1. Go back to init.jsp
2. Open snippet4.txt, copy the code in section 1 and append it at the end of the imports.
3. Now, open view.jsp.
4. Copy the code in section 2 of snippet4.txt and append it at the end of the jsp.

The portlet now displays a message like this: *(Figure 6.16, page 219)*

Figure 6.16:




## 6.5.5 Categorizing the configuration

To make it easier for platform administrators to find the right configuration options, Liferay provides a mechanism for developers to specify a category in which the configuration will be shown in the auto-generated System Settings UI in the Control Panel.

By default, the following configuration categories are defined:

- Web Experience Management
- Collaboration
- Productivity
- Platform
- Other: You must use the `@ExtendedObjectClassDefinition` annotation as in the following example:

```
@ExtendedObjectClassDefinition(category = "platform")
@Meta.OCD(
    factory = true,
   id = "com.liferay.portal.ldap.configuration.SystemLDAPConfiguration",
    localization = "content/Language"
)
```

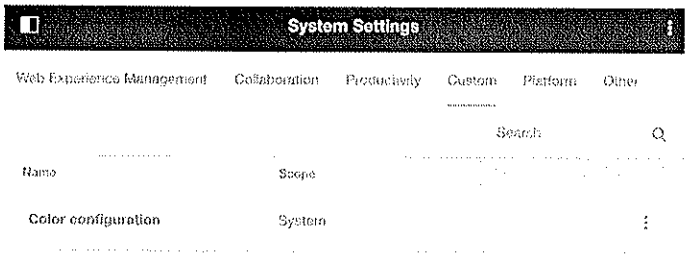
### Exercise: Categorising the configuration

1. **Open** the configuration class in eclipse.
2. **Copy** the code in section 1 of **snippet5.txt** and **paste** it after the imports.
3. **Open** `build.gradle`, **copy** the code in section 2 of `snippet5.txt`, and **paste** it in the dependencies section.
4. **Right-click** on the `project` → `Gradle` → `Refresh Gradle Project` to update the project.
5. **Go back** to the configuration: *Control Panel* → *Configuration* → *System Setting*, you will find a new '`custom`' category created and you will be able to find your configuration under the newly-created category. *(Figure 6.17, page 221)*

## 6.5.6 Support different configuration per instance, Site, or portlet

It's common to need different configurations depending on the scope of an application. Liferay 7 provides an easy way to leverage a new framework, called the Configuration API, that is based on the standard OSGi Configuration Admin API, which we've already seen.

Figure 6.17:




## Using the Configuration Provider

Instead of receiving the configuration directly, we need to retrieve it through the `ConfigurationProvider` and create a class implementing `ConfigurationBeanDeclaration`. Here is an example:

```
@Component
    public class RSSPortletInstanceConfigurationBeanDeclaration
        implements ConfigurationBeanDeclaration {

        @Override
        public Class getConfigurationBeanClass() {
            return RSSPortletInstanceConfiguration.class;
        }

    }
```

We can now use a `ConfigurationProvider` by getting a reference to it. Examples:

- For components:

```
@Reference
protected void setConfigurationProvider(ConfigurationProvider
configurationProvider) {
    _configurationProvider = configurationProvider;
}
```

- For Service Builder services:

```
@ServiceReference(type = ConfigurationProvider.class)
protected ConfigurationProvider configurationProvider;
```

We can now use all the corresponding methods to access the right scope.

- `getCompanyConfiguration()`
- `getGroupConfiguration()`
- `getPortletInstanceConfiguration()`

### Real-world example

```
JournalGroupServiceConfiguration configuration =
    configurationProvider.getGroupConfiguration(
        JournalGroupServiceConfiguration.class, groupId);

MentionsGroupServiceConfiguration configuration =
  _configurationProvider.getCompanyConfiguration(
    MentionsGroupServiceConfiguration.class, entry.getCompanyId());
```

## 6.5.7 Accessing the Portlet Instance Configuration Through the PortletDisplay

Often it's necessary to access a portlet's settings from its JSPs or from Java classes that are not OSGi components. To make it easier to read the settings in these cases, a new method has been added to `PortletDisplay`.

```
RSSPortletInstanceConfiguration rssPortletInstanceConfiguration =
    portletDisplay.getPortletInstanceConfiguration(
        RSSPortletInstanceConfiguration.class);
```

## 6.5.8 Specifying the scope of the configuration

Besides the category, the ExtendedObjectClassDefinition annotation allows you to specify the scope of the configuration too. The valid options are:

- Scope.GROUP: for Site scope
- Scope.COMPANY: For virtual instance scope
- Scope.SYSTEM: for system scope

Here is an example:

```
@ExtendedObjectClassDefinition(
   category = "productivity", scope = ExtendedObjectClassDefinition.
    Scope.GROUP
)
@Meta.OCD(
   id = "com.liferay.dynamic.data.lists.form.web.configuration.
    DDLFormWebConfiguration",
   localization = "content/Language", name = "%ddl.form.web.configuration.
    name"
)
public interface DDLFormWebConfiguration {
...
}
```

### 6.5.9 Implementing Configuration Actions

We have just learned the flexible mechanism for configuration applications. Now, let's see how we can implement configuration actions for your application. The configuration action is invoked when a User clicks on the gear icon of a Liferay application and selects *Configuration*.

*(Figure 6.18, page 224)*

### 6.5.10 Implementing Configuration Actions - steps

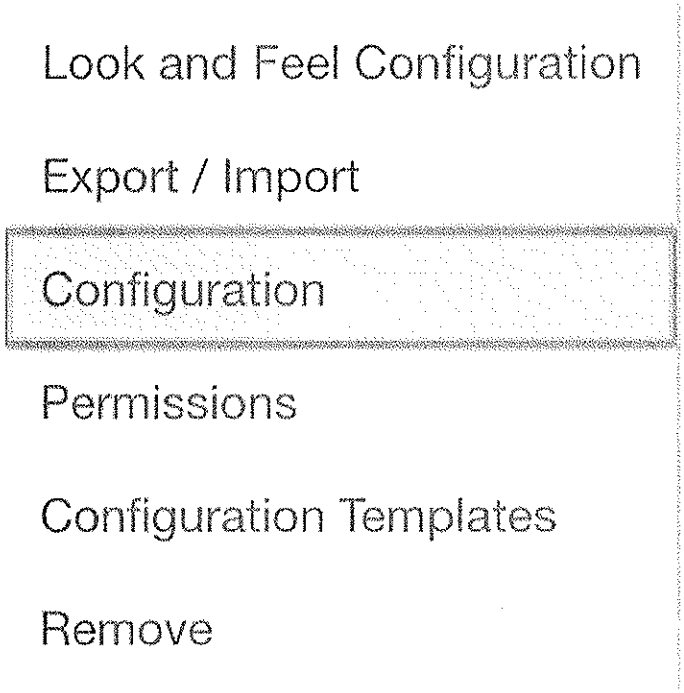
When a User clicks on a Liferay application's gear icon and selects Configuration, the application's *configuration* action is invoked.

Liferay applications support a default configuration action. If you click on the gear icon of a Liferay application that has not been customized and then select Configuration, you'll find two standard tabs: Permissions and Sharing. These tabs provide standard options for configuring who can access your application and how you can make your application more widely available.

Now we will implement a custom tab to allow custom configuration fields. To implement a configuration action, follow these steps:

1. **Create** an interface to represent your configuration.
2. **Implement** your application class and **add** a reference to your configuration in your application class.

Figure 6.18:




3. **Implement** your configuration action class and **add** a reference to your configuration in your configuration action class.
4. **Implement** the User interface for configuring your application.

### 6.5.11 Creating a configuration interface

In order to save time, we will reuse the `com.example.api.ColorConfiguration` that we have created previously.

In the real world, it is preferable to create different configuration interfaces for application configuration and configuration action.

### 6.5.12 Referencing the configuration from the application class

This step is exactly the same as we have seen previously. For the exercise, we are going to reuse the same portlet class, so the reference is already included.

## Exercise: Implementing a configuration action class

1. **Create** a class `com.example.impl.ColorConfigurationAction`.
2. **Copy** the content of **snippet6.txt** and **replace** the class code with it.

Note that we specify `configurationPid`, `configurationPolicy` and `javax.portlet.name` to the component.

And we override the `processAction` method so that it reads a URL parameter from the action request.

We also override the `include` method so that it sets the configuration as an attribute of the HTTP servlet request, and then invokes the include method of the `BaseJSPSettingsConfigurationAction` class.

## Exercise: Implementing the User Interface for the Application Configuration

1. **Open** *init.jsp*.
2. **Copy** the content of *snippet7.txt* and replace the code of *init.jsp* with it.
3. **Open** *view.jsp*.
4. **Comment out** the scriptlet.
5. **Copy** the content of *snippet8.txt* and paste it and the end of *view.jsp*.

In **init.jsp**, the scriptlet at the end of the file uses one of these variables, renderRequest, to get the configuration which was stored in the renderRequest by your portlet's doView method. Finally, the value of a specific field (favoriteColor) is read from the configuration.

**view.jsp** simply checks whether or not the favoriteColor variable is empty. If it's empty, a message is displayed that tells the User to select a favorite color in the portlet's configuration. If the favoriteColor variable is not empty, the name of the selected color is displayed in the selected color.

## Exercise: Implementing the User Interface for the Application Configuration

1. **Create** a file named `configuration.jsp` in the same folder of `init.jsp` and `view.jsp`

2. **Copy** the content of *snippet9.txt* and **paste** it into the newly-created file.

This JSP uses the `<liferay-portlet:actionURL/>` and `<liferay-portlet:renderURL` tags to construct two URLs in the variables `configurationActionURL` and `configurationRenderURL`. The JSP presents a simple form that allows the User to select a favorite color. When the User submits the form, the `configurationActionURL` is invoked and the application's `processAction` method is invoked with the `favoriteAction` included as a request parameter.

If the request fails, the User is redirected to the configuration page.

### Exercise: Implementing the User Interface for the Application Configuration

Now **click** on the gear icon of the portlet, and select *Configuration*. Select a favorite color and click Save. You can see your new configuration is active.

*(Figure 6.19, page 226)*

Figure 6.19:


## 6.6 LDAP

- LDAP (Lightweight Directory Access Protocol) is an application protocol for accessing and maintaining distributed directory information services over an IP network.
- LDAP servers process queries and updates to LDAP directories. Whereas databases are designed to process a high volume of updates in a short timespan, LDAP directories are optimized for read performance.
- Many organizations use LDAP directories to store information such as employee credentials, organization hierarchies, HR data, customer contact information, and network information.

- Many web applications support LDAP-integration and many organizations use credentials stored in LDAP to provide access a variety of web applications. In this section, we'll enable LDAP users to authenticate to Liferay.

## 6.6.1 Liferay LDAP integration

Liferay provides three primary levels of integration for LDAP:

1. LDAP Authentication
2. LDAP User Import
3. LDAP User Export

These services are encapsulated in the portal-ldap module. `com.liferay.portal.ldap.jar`.

## 6.6.2 LDAP Authentication

LDAP authentication is unlike other SSO implementations within the platform (e.g. Token, CAS, OpenSSO, etc). LDAP authentication uses a lower level Liferay concept of an Authenticator. Authenticators are injected into the Liferay AuthPipeline and used as part of the platform's own authentication process. Other SSO implementations use a bypass mechanism (AutoLogin) which bypasses the platform's authentication process.

### Implementation

The implementation is encapsulated inside of `com.liferay.portal.ldap.internal.authenticator.LDAPAuth`. LDAPAuth is an Authenticator component that is registered to the authentication pipeline.

LDAPAuth is configured via 3 sets of configuration properties:

1. LDAPAuthConfiguration
2. LDAPServerConfiguration
3. SystemLDAPConfiguration

## 6.6.3 LDAP User Import

Liferay has a generic User import service which imports users from any external system into Liferay's User store. The LDAP User import is a LDAP specific implementation of this service.

There are two paths during which Users can be imported from LDAP:

1. When LDAP authentication is enabled, any User that successfully authenticates into the portal will be automatically imported into the Liferay User store.
2. When LDAP scheduled import is enabled (via `LDAPImportConfiguration.enabled`), then on a regularly configured interval (`LDAPImportConfiguration.importInterval`), Users are imported from LDAP into the Liferay User store.

### Implementation

`com.liferay.portal.ldap.internal.exportimport.LDAPUserImportImpl` provides the bulk of the implementation for the service. It communicates with LDAP to retrieve User and group information. `LDAPUserImportImpl` uses a `LDAPToPortalConverter` to convert from LDAP objects and attributes to Liferay objects (`User`, `Contact`, `Group`, `Role`).

LDAPUserImportImpl is configured via

1. LDAPImportConfiguration
2. LDAPServerConfiguration

## 6.6.4 LDAP User Export

Liferay has a generic User export service that complements the user import service. The User export service is capable of exporting users to any external system from Liferay's User store. The LDAP User export is a LDAP specific implementation of this service.

There are a few conditions under which users would be exported:

- When users are added to Liferay's User store
- When User profiles are modified in Liferay's User store
- When users are added to User groups
- When users are removed from User groups

- When users are added to roles
- When users are removed from roles

### Implementation

`com.liferay.portal.ldap.internal.exportimport.LDAPUserExportImpl` provides the bulk of the implementation for the service. It communicates with LDAP to store User profile information. `LDAPUserExportImpl` uses a `PortalToLDAPConverter` to convert from Liferay objects (User, Contact) to LDAP objects.

`LDAPExportUserImpl` configured via:

1. LDAPExportConfiguration
2. LDAPServerConfiguration

To trigger export, the implementation uses ModelListeners:

- Exporting User information when User and contact information changes:
  - `com.liferay.portal.ldap.internal.exportimport.model.listener.UserModelListener`
  - `com.liferay.portal.ldap.internal.exportimport.model.listener.ContactModelListener`
- Exporting User information when UserGroup memberships change for users
  - `com.liferay.portal.ldap.internal.exportimport.model.listener.UserGroupModelListener`
- Exporting User information when Role membership changes for users
  - `com.liferay.portal.ldap.internal.exportimport.model.listener.UserModelListener`

## 6.6.5 Configuration

Prior to 7.0, LDAP-related configurations were present in a couple of places:

- portal.properties
- database

- passwordPolicyEnabled
- Set this to true to use LDAP's password policy instead of the portal password policy. If set to true, it is possible that portal generated passwords will not match the LDAP policy. See the "passwords.regexptoolkit.*" properties in portal.properties for details on configuring RegExpToolkit in generating these passwords.

## LDAPExportConfiguration

Note: None of the values for LDAPExportConfiguration will be used unless `LDAPAuthConfiguration.enabled` has been set to true.

- exportEnabled
- Setting for exporting users from the portal to LDAP
- exportGroupEnabled
- Setting for exporting groups and their associations from the portal to LDAP

## LDAPImportConfiguration

Note: None of the values for LDAPImportConfiguration will be used unless `LDAPAuthConfiguration.enabled` has been set to true.

## LDAPServerConfiguration

Notes for specific properties:

- authSearchFilter
- Active Directory stores information about the User account as a series of bit fields in the UserAccountControl attribute. If you want to prevent disabled accounts from logging into the portal you need to use a search filter similar to the following: `(&(objectclass=person)(userprincipalname=@email_address@)(!(UserAccountControl:1.2.840.113556:1.4.803:=2)))`

  See the following links: `http://support.microsoft.com/kb/305144/` `http://support.microsoft.com/?kbid=269181`

### SystemLDAPConfiguration

Notes for specific properties:

- pageSize
- Set the page size for directory servers that support paging. This value needs to be 1000 or less for the Microsoft Active Directory Server.
- rangeSize
- Set the number of values to return in each query to a multivalued attribute for directory servers that support range retrieval. The range size must be 1000 or less for Windows 2000 and 1500 or less for Windows Server 2003.
- connectionProperties
- To enable LDAP connection pooling, you have to set `com.sun.jndi.ldap.connect.pool` to true and provide additional JVM system properties via the JVM start-up options via:
- `java ... -Dcom.sun.jndi.ldap.connect.pool. maxsize=50 -Dcom.sun.jndi.ldap.connect. pool.timeout=10000`

  See the following link:

  `http://docs.oracle.com/javase/6/docs/technotes/guides/jndi/jndi-ldap.html`

## 6.6.6 LDAP Exercise

### Start server

For our exercise, we are using a pre-configured Java-based LDAP Server called OpenDJ. You can find more information of the OpenDJ from their website: http://opendj.forgerock.org/

First you have to **start** the LDAP server.

Goto: `~/servers/OpenDJ` directory with command:

```
cd ~/servers/OpenDJ
```

and **start** server with following command:

### 6.6.8 Bonus Exercise

*Use the FakeSmtp server and use Liferay to reset the User password. Verify that the password has also been changed in LDAP.*

*To do that, use the control panel to see example Alice's userPassword field value and copy that, then request a new password, change it, and verify that the field value has been changed.*

*Second, create a new user through Liferay and see if the user information has been exported to LDAP.*

## 6.7 SSO Autologins

There are several different ways to integrate with SSO providers.

- Token
- Facebook
- OpenID
- CAS
- OpenSSO
- NTLM
- Google
- Only one SSO provider is supported per deployment. See LPS-52940

### 6.7.1 Auto-login

- In 6.2, custom auto-logins are registered in `portal-ext.properties` as comma delimited class names that implement `com.liferay.portal.kernel.security.auto.login.AutoLogin` in property: `auto.login.hooks`
- In 7.0, they're registered as OSGi @Component-s. Old auto logins are modularized and new auto logins can be implemented with the following pattern:

```
.java
@Component(
    configurationPid = "...Configuration",
   configurationPolicy = ConfigurationPolicy.OPTIONAL, immediate = true,
    service = AutoLogin.class
)
public class ExampleAutoLogin extends BaseAutoLogin {

}
```

## 6.7.2 Token-based autologin

Token SSO authentication was introduced in Liferay 7 to standardize support for Shibboleth and SiteMinder, and any other SSO product which works on the basis of propagating a token via one of the following mechanisms:

- HTTP request parameter
- HTTP request header
- HTTP cookie
- Session attribute

The token contains either the Liferay Portal user's screen name or email address, whichever Liferay Portal has been configured to use as a username for the particular company (portal instance).

### Configuration location:

- UI instance level: `Control Panel` → `Configuration` → `System Settings` → `Foundation` - find `Token Based SSO`
- UI system level: `Control Panel` → `Configuration` → `System Settings` → `Foundation` → `Token Based SSO`
- From configuration file: `{liferay-home}/osgi/configs/com.liferay.portal.security.sso.token.configuration. TokenConfiguration.cfg`

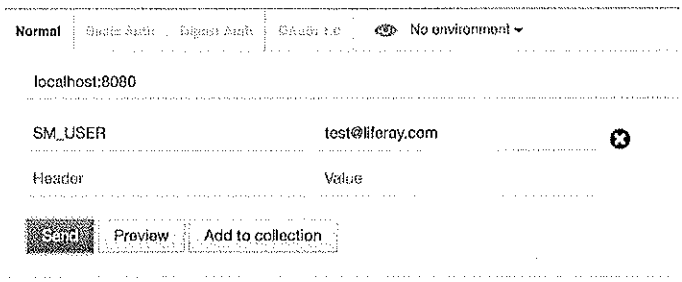
### 6.7.3 Token-based autologin exercise:

This exercise requires that you have the `postman` Chrome extension installed or similar, which allows you to insert HTTP headers for the request.

- Go to `Control Panel` → `Configuration` → `System Settings` → `Foundation` → find `Token Based SSO` and click and choose `Edit`.
- Click `Save`.
- Use Postman chrome plugin to add new http header `SM_USER` as pictured below.
- Click `Send`.

*(Figure 6.22, page 238)*

Figure 6.22:




Now your Chrome should be logged in.

### 6.7.4 CAS Single Sign On Authentication

At present, if CAS SSO authentication is enabled, this will override all other SSO implementations and take full control over the Liferay Portal login and logout functionality. This limitation has been logged as issue LPS-52940.

#### Managing CAS SSO authentication

Configuration can be applied at two levels:

- System (all portal instances)

- Portal instance

To configure the CAS SSO module at the System level, navigate to Liferay's Control Panel, click on Configuration Admin, and find the CAS module.

The values configured here provide the default values for all portal instances.

## Configuration location:

- UI instance level: `Control Panel` → `Configuration` → `Instance settings` → `Authentication` → `CAS SSO`
- UI system level: `Control Panel` → `Configuration` → `System Settings` → `Foundation` → `CAS SSO`
- Configuration file: `{liferay-home}/osgi/configs/com.liferay.portal.security.sso.cas.module.configuration.CASConfiguration.cfg`

## The configuration fields:

- `importFromLDAP`: Users authenticated from CAS that do not exist in the portal are imported from LDAP. LDAP must be enabled separately.
- `loginURL`: CAS server login URL
- `logoutOnSessionExpiration`: If true, then browser with expired sessions will be redirected to the CAS logout URL
- `logoutURL`: The CAS server logout URL. Set this if you want the portal's logout function to trigger a CAS logout.
- `serverName`: The name of the platform (e.g., liferay.com). If the provided name includes the scheme (https://, for example), then this will be used together with the path/c/portal/login to construct the URL that the CAS server will provide tickets to. If no scheme is provided, then the scheme normally used to access the Liferay login page will be used.
- `serverURL`: The CAS server base URL. For example `http://server.com:8080/cas-web`

- `serviceURL`: If provided, this will be used as the URL that the CAS server will provide tickets to. This overrides any URL constructed based on Server Name as above.

- `noSuchUserRedirectURL`: Set the URL to redirect the User if the User can authenticate with CAS but cannot be found in the platform. If "Import from LDAP" is enabled, the User is redirected if he cannot be found or could not be imported from LDAP.

## 6.7.5 Facebook Connect Single Sign On Authentication

Facebook SSO is an integration with Facebook's Graph API and works on the basis of retrieving the User's Facebook profile information, and then attempting to match existing Liferay users on either facebook ID or email address.

In order to integrate Liferay with Facebook, you must first create an "application" on Facebook's website at `https://developers.facebook.com`. This is because Facebook Connect requires Liferay to authenticate using the OAuth 2.0 protocol. Facebook will provide you with the necessary application ID and secret which will be used in OAuth messages sent between Liferay and Facebook. One benefit of this is allowing the Facebook User to revoke access from Liferay at a later date.

The integration will attempt to retrieve the following Facebook profile information in order to successfully create a User on the Liferay Portal:

- Email Address
- First Name
- Last Name
- Gender

### Managing Facebook Connect SSO Authentication

Configuration can be applied at two levels:

- System (all platform instances)
- Platform instance