Onboard插件编程指南

# 概述

本篇指南将为你介绍如何从零开始实现一个Onboard插件

实现一个onboard插件需要完成两部分工作，分别为插件的前端和插件的后端。后端需要完成插件所需的领域模型、数据访问接口、业务类以及通过Controller实现的Rest API。前端需要完成插件所需的前端页面、描述样式的less文件以及描述页面交互逻辑的js文件。

# 插件后端

在编写onboard插件后端之前，我们先来看看一个插件示例的目录结构。

## 目录结构

├─pom.xml

├─osgi.bnd

├─src

│ └─main

│ ├─java

│ │ └─com

│ │ └─onboard

│ │ └─plugin

│ │ ├─sample

│ │ │ ├─activity

│ │ │ ├─api

│ │ │ ├─impl

│ │ │ ├─mapper

│ │ │ ├─model

│ │ │ ├─dto

│ └─resources

│ ├─com

│ │ └─onboard

│ │ └─plugin

│ │ └─sample

│ │ └─mapper

│ ├─META-INF

│ │ └─spring

│ └─WEB-INF

**Pom.xml**文件用来管理插件所需的第三方依赖，**osgi.bnd**管理plugin bundle中需要引入以及暴露的package，需要引用的Bundle等。

**Src**目录中，com.onboard.plugin.sample中需要定义插件所需的业务接口。在其子package impl中完成接口的实现。

子package **activity**中定义了当前业务所需要的回顾信息的生成方法。

子package **model**中定义了插件所需的领域模型。

子package **mapper**中定义了插件所需的数据访问接口。

子package **dto**中定义了传输所需的DTO类。

子package **api**中定义了插件所需的Controller来提供Rest API。

**Resource**文件夹中，**com.onboard.plugin.sample.mapper**中存放了mapper包中接口对应的**mybatis**配置文件。

**META-INF/spring**文件夹中一般存放plugin.sample.**aop**.xml, plugin.sample.**osgi**.xml, plugin.sample.**git**.xml这三个文件。分别用来进行切面配置管理，osgi服务管理以及spring bean的声明。

**WEB-INF**文件夹中存放web相关的配置，包括**applicationContext**.xml, **applicationOSGI**.xml, **applicationSecurity**.xml, **dispatcher-servlet**.xml, **web**.xml文件。分别进行Bean的声明，OSGI服务的引用，安全认证的管理，web相关配置等。

## 建立项目、引入第一方和第三方依赖

在写一个插件之前，我们需要建立一个maven项目，在项目根目录下创建一个pom.xml文件。

在其中，将parent设为onboard.parent，并定义插件的artifactId：

<parent>

<groupId>com.onboard</groupId>

<artifactId>com.onboard.parent</artifactId>

<version>3.0.0-SNAPSHOT</version>

<relativePath>../../pom.xml</relativePath>

</parent>

<packaging>bundle</packaging>

<artifactId>com.onboard.plugin.sample</artifactId>

<name>com.onboard.plugin.sample </name>

<description>com.onboard.plugin.sample </description>

接下来，引入插件所需的第一方和第三方依赖，比如我们需要使用onboard中的回顾生成工具来生成回顾并需要spring security来做认证：

<dependencies>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>org.springframework.security.config</artifactId>

</dependency>

<dependency>

<groupId>org.springframework.security</groupId>

<artifactId>org.springframework.security.web</artifactId>

</dependency>

<dependency>

<groupId>com.onboard</groupId>

<artifactId>com.onboard.service.activity</artifactId>

<version>${project.version}</version>

</dependency>

</dependencies>

## 设计数据库、使用自动化工具生成mapper和数据库映射实体类

（待补充）

## 设计领域模型

领域模型（domain model）是描述业务用例实现的对象模型。在onboard中领域模型有几种类型：

Attachable: 可添加附件的对象。

Broadable: 可以进看板的对象。

Commentable：可以评论的对象。

Identifiable: 具有id和类型的对象。

Indexable: 可以被索引的对象。

Iterable: 可以进入迭代的对象。

Subscribable：可以被订阅的对象。

Taggable： 可以打标签的对象。

每种领域模型都可以通过实现接口来成为上述一种或几种类型的对象，每个领域模型一般都会继承数据库映射实体类。

插件开发者可以根据需求来实现自己的领域模型，放入com.onboard.plugin.\*.model中。

## 完成业务逻辑

首先在com.onboard.plugin.\*中定义一个业务接口。

在com.onboard.plugin.\*.impl中定义实现类，并通过@Service标注的方式装配Bean。

在META-INF/spring中plugin.git.xml中加入配置来支持Bean的自动装配:

<context:component-scan base-package=*"com.onboard.plugin.sample"*/>

### 生成回顾

如果业务的某个表示创建、更新的方法需要生成activity，我们在实现业务类时需要完成一些额外的工作。

首先操作方法的目标类需要是一个Identifiable类。

其次Service的实现需要继承DefaultIdentifiableService或实现IdentifiableService接口。

然后需要在META-INF/spring文件夹中的plugin.xxx.osgi文件中声明一个OSGI服务来让Kernel监听到这个服务：

<service id="xxxService" interface="com.onboard.plugin.xxx.XxxService, com.onboard.service.common.identifiable.IdentifiableService" ref="xxxServiceBean" />

最后需要在com.onboard.plugin.xxx.activity包中完成Activity的生成类，并实现ActivityGenerator接口，并将其装配到Bean当中，并声明OSGI服务。

<service id=*"xxxActivityGenerator"* interface=*"com.onboard.service.activity.ActivityGenerator"* ref=*"xxxActivityGeneratorBean"* />

### 生成索引

生成索引与生成回顾类似，需要完成一些额外工作。

首先领域模型需要实现Indexable接口。

其次在com.onboard.plugin.xxx.index中建立类来需要实现IndexableService接口。

然后需要在META-INF/spring文件夹中的plugin.xxx.osgi文件中声明一个OSGI服务来让Kernel监听到这个服务：

<service id="xxxService" interface="com.onboard.plugin.xxx.XxxService, com.onboard.service.index.custom. IndexableService" ref="xxxServiceBean" />

最后声明OSGI服务，来注册到kernel中:

interface=*"com.onboard.service.index.custom.IndexableService"* ref=*"xxxIndexableServiceBean"* />

### 生成通知

生成通知时，首先需要在com.onboard.plugin.xxx.notification.rule包中实现类继承抽象类SimpleNotificationRule来定义通知条件。

其次在com.onboard.plugin.xxx.notification.email中实现类继承抽象类AbstractEmailNotification，来定义通知模板。

最后在META-INF/spring中的plugin.xxx.osgi.xml中声明服务来让kernel监听到这个服务:

<service id=*"xxxNotification"* interface=*" com.onboard.service.notification.NotificationMethod"* ref=*"xxxNotificationBean"* />

<service id=*"xxxNotificationRule"* interface=*" com.onboard.service.notification.NotificationRule"* ref=*"xxxNotificationRuleBean"* />

## 完成REST API

### 完成Security的配置

Onboard实现了前后端的分离，前端服务器和后端服务器使用BASIC AUTH的方式进行验证。

在plugin中需要完成Security的配置，在resource文件夹的WEB-INF中完成applicationSecurity.xml文件，在其中声明myEncoder和userDetialsServiceBean两个Bean。

<beans:bean id=*"myEncoder"* class=*"com.onboard.service.security.auth.CustomizedPasswordEncoder"*/>

<beans:bean id=*"userDetailsServiceBean"* class=*"com.onboard.service.security.auth.UserAuthenticationService"*/>

定义需要进行拦截的url特征：

<http auto-config=*"true"* use-expressions=*"true"* create-session=*"stateless"*>

<intercept-url pattern=*"/\*\*"* access=*"isAuthenticated()"* />

<http-basic />

</http>

声明authentication-manager：

<authentication-manager>

<authentication-provider user-service-ref=*"userDetailsServiceBean"*>

<password-encoder ref=*"myEncoder"*>

<salt-source user-property=*"salt"*/>

</password-encoder>

</authentication-provider>

</authentication-manager>

### 完成web.xml和dispatcher-servlet.xml的配置

参照web.api bundle完成。

### DTO

在com.onboard.plugin.xxx.dto包中实现传输用的类。其中仅仅包含数据库中的字段和Getter setter方法。

### 完成Controller

在com.onboard.plugin.xxx.api包中实现相关Controller

在osgi.bnd中可以设置plugin的uri的header，onboard插件统一命名规则为:

{host}:{port}/plugins/${pluginName}/api/\*\*\*

Controller中我们推荐使用Rest风格的URL设计。

# 插件前端

前端插件的编写主要使用了AngularJS框架，并通过Theamleaf将插件所有的JavaScript文件、Less/Css文件以及Html文件引入Onboard之中。

## 目录结构

├─src

└─main

└─resources

├─static

│ ├─js

│ │ └─ng-modules

│ │ └─plugin

│ │ └─插件js文件

│ └─less

│ └─插件less文件

└─templates

└─plugin

└─插件html文件

## JaveScript

不同插件所使用的Js文件不允许出现同名，否则会产生覆盖的情况。

任意一个插件注册时需要在主要的Js文件中书写一下语句，以将插件的Module注入到Onboard的Module中：

angular.module('onboard').requires.push('pluginName');

angular.module('pluginName',['plugin'])

.run(['pluginService', **function**(pluginService) {

// 通过PluginService内方法将插件入口写入Onboard

}]);

根据插件入口类型的不同，一共有7种不同的入口，分别为：

pluginService.registerDrawer方法，插件为右边展开的Drawer

其参数需要包含有：

{

name: 'drawerSample', // 插件名

template: 'drawerSample.html' // 插件模板

size: 'drawer-fl' // 插件窗口大小（可选）

}

pluginService.registerTab方法，插件为Drawer页面的Tab

其参数需要包含有：

{

name: 'tabSample', // 插件名

title: '样例TAB', // 插件所展示的Tab名称

templateUrl: 'tabSample.html' // 插件模板

}

pluginService.registerCompanyPlugin方法，插件为团队层面的插件，展示于团队选择页面Header

其参数需要包含有：

{

title: '团队层-样例插件', // 插件名称

'ui-sref': 'companyPluginSample' // 插件页面路由

}

pluginService.registerProjectPlugin方法，插件为项目层面的插件，展示于项目选择页面Header

其参数需要包含有：

{

title: '项目层-样例插件', // 插件名称

'ui-sref': 'projectPluginSample' // 插件页面路由

}

pluginService.registerSidePlugin方法，插件为项目层面的插件，展示于项目页面左侧边栏

其参数需要包含有：

{

icon: 'fa-plug', // 标签图标

title: '项目插件', // 插件名称

'ui-sref': 'sidebarPluginSample' // 插件页面路由

}

pluginService.registerAccountPlugin方法，插件为个人设置层面的插件，展示于个人设置页面

其参数需要包含有：

{

title: '个人设置-样例插件', // 插件名称

'ui-sref': 'account.accountPluginSample' // 插件页面路由

}

pluginService.registerFooterPlugin方法，插件为项目层面的插件，展示于项目页面左侧边栏下方

其参数需要包含有：

{

icon: 'fa-plug', // 标签图标

title: '项目插件', // 插件名称

'ui-sref': 'footerPluginSample' // 插件页面路由

}

其他JavaScript内容的书写遵循AngularJS规范即可使用。

## Less/Css文件

将Less/Css文件放入对应的文件即可。

## Html文件

在插件中Html文件使用Theamleaf渲染进主模板Onboard.html。故需要在插件的Html文件中通过th:fragment属性的方式将插件内容引入，其代码为：

<script th:fragment=*"plugin"* type=*"text/ng-template"* id=*"pluginName"*>

<div> … </div>

</script>

其中带有th:fragment=*"plugin"* 属性的元素会被引入至Onboard.html。