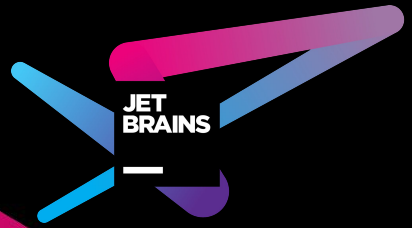


Краш-курс по IntelliJ IDEA Plugin DevKit

Юрий Артамонов

Автор ты кто

Юрий Артамонов @jreznot

- Разрабатывал фреймворки и библиотеки для Java на протяжении 9 лет
- Грустил от их несовершенной поддержки в IDE
- Недавно пришёл в команду IntelliJ IDEA

План действий

1. Система плагинов IntelliJ IDEA
2. Моделируем Java фреймворк
3. Пишем к нему полезный плагин
4. Когда это делать не нужно?
5. Куда копать дальше?

Мотивация

- У всех есть свои наработки: in-house фреймворки, библиотеки, конфиги, процессы и соглашения
- Рутинная работа, связанная с исходным кодом, проектом, тестами, развёртыванием

Как бороться с рутиной

- Делать больше фреймворков!
- Генерировать больше кода!
- **Улучшать свой основной инструмент - IDE**

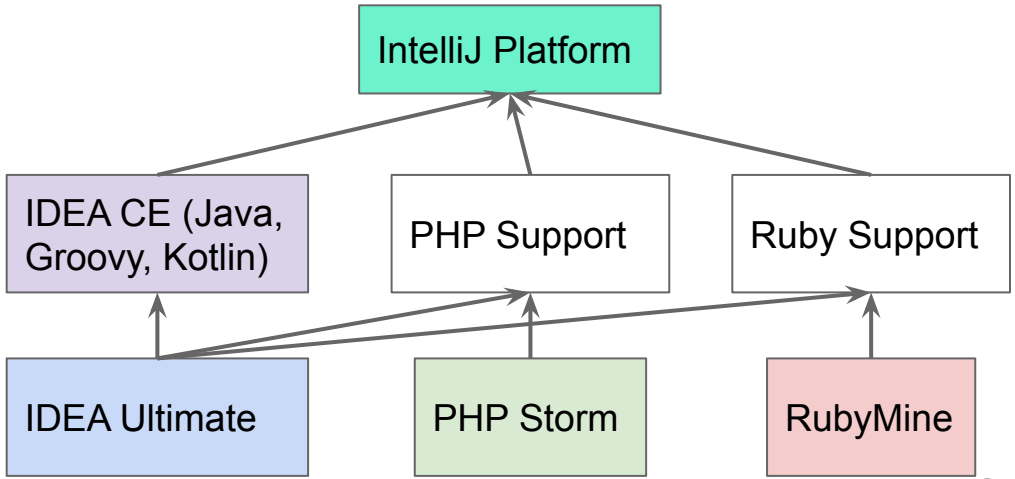
Делимся опытом!

Возможности плагинов

- ★ Языки и форматы файлов
- ★ Статический анализ кода на лету
- ★ Навигация по коду
- ★ Улучшения редактора
- ★ Рефакторинг и генерация кода
- ★ Взаимодействие с инструментами и сервисами
- ★ Миграция старого кода на новый API

Ключевые внутренности IDE

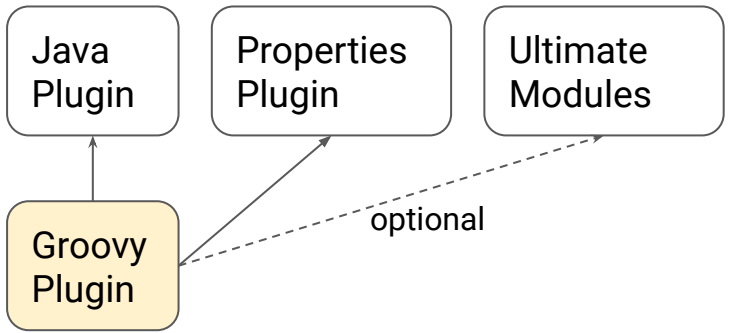
- Компоненты и сервисы
- Виртуальная файловая система (VFS)
- Поддержка языков (PSI)
- Редактор кода
- Инспекции
- Индексы
- Фоновые процессы
- UI библиотека
- Точки расширения



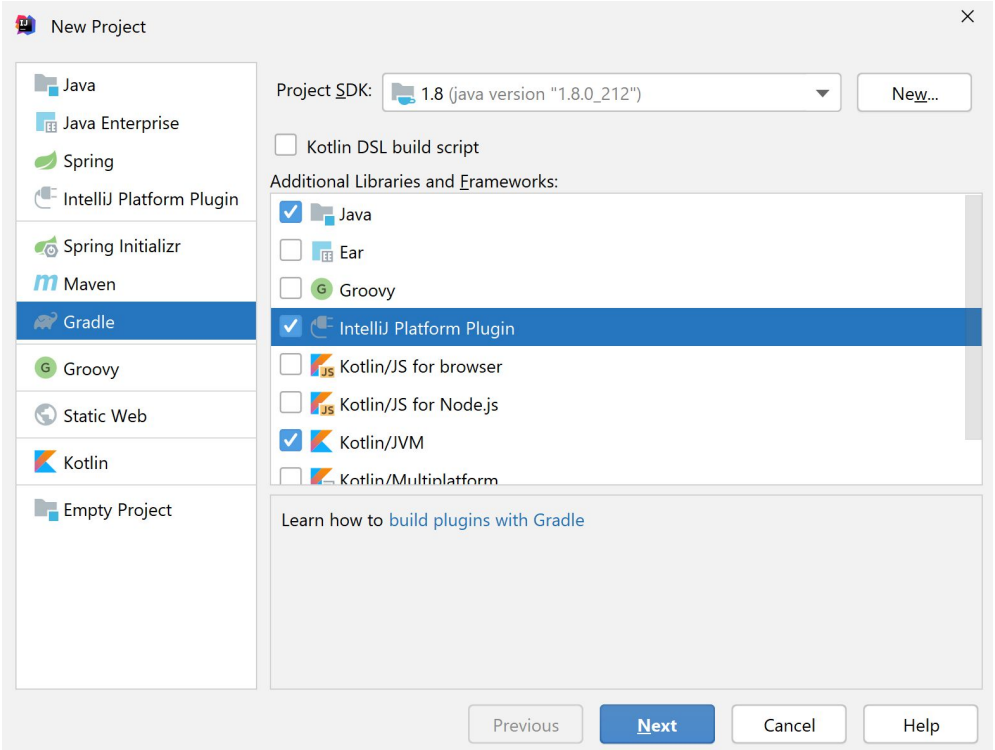
Система плагинов

Все JetBrains IDE состоят из плагинов:

1. Классы плагина загружаются отдельным загрузчиком классов
2. Каждый плагин реализует точки расширения IDE
   и может объявлять свои
3. Плагины могут зависеть от модулей IDE и от других плагинов



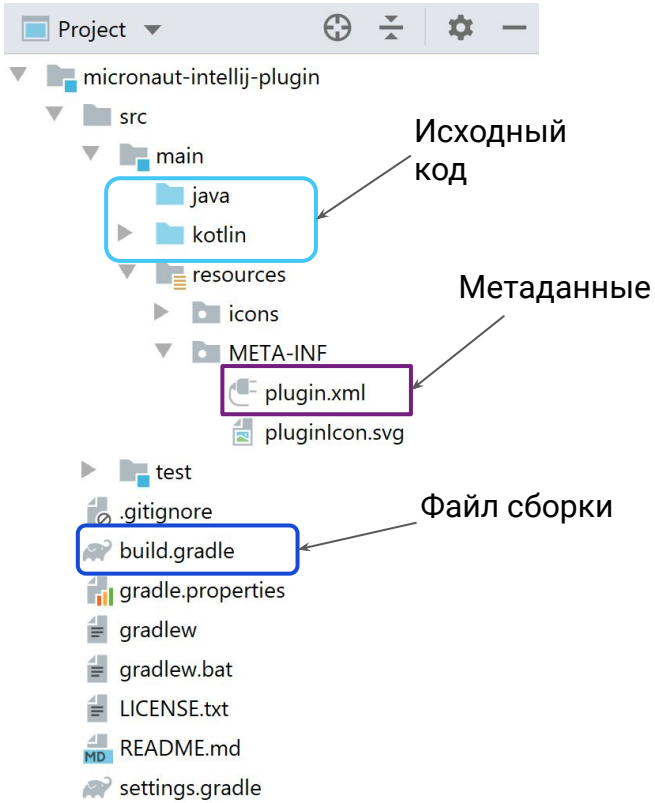
Как начать



1. Проверить/включить Plugin DevKit
2. Создать Gradle проект с библиотекой IntelliJ Platform Plugin
3. Объявить и реализовать нужные точки расширения

Рекомендуем Kotlin!

Структура плагина



.../resources/META-INF/plugin.xml

```
<idea-plugin>
   <id>micronaut-intellij-plugin</id>
   <name>Micronaut Framework Support</name>
   <description><![CDATA[
   Provides support for Micronaut Framework for JVM languages
]]></description>

   <depends>com.intellij.modules.java</depends>

   <extensions defaultExtensionNs="com.intellij">
       <!-- Add your extensions here -->
   </extensions>

   <actions>
       <!-- Add your actions here -->
   </actions>
</idea-plugin>
```

Точки расширения

- Огромное множество:
  action, inspection, intention action, reference contributor, ui settings group, language parser, language formatter и т.д.
- Регистрируются в XML-файле –
  дескрипторе плагина plugin.xml
- Полный список: *LangExtensionPoints.xml, PlatformExtensionPoints.xml, VcsExtensionPoints.xml*, etc.

Пример: поддержка Groovy – плагин (plugin.xml > 1300 строк)

github.com/JetBrains/intellij-community

Micronaut - вид сбоку

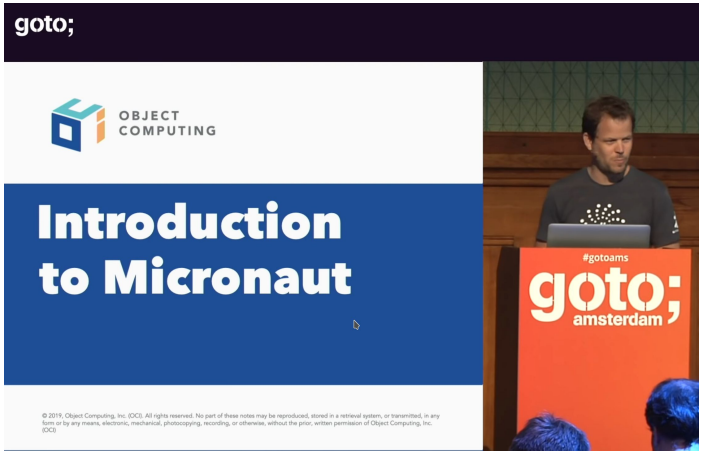
Фреймворк для построения легковесных модульных приложений на JVM (Java / Kotlin / Groovy)

Возможности:

- Dependency Injection
- HTTP Web Services (Sync/Async)
- HTTP Client
- Data Repositories
- Compile-time everywhere
- AOT Compatible

Что посмотреть

1. Graeme Rocher - **Introduction to Micronaut**
2. Кирилл Толкачёв, Максим Гореликов - **Micronaut vs Spring Boot, или Кто тут самый маленький?**



Что тут в вашем фреймворке поддерживать?

Наносим непоправимую пользу

Проблема:

1. Создали проект
2. Запустили
3. ~~Profit~~ Ничего не происходит!

Документация: If you are using Java or Kotlin and IntelliJ IDEA make sure you have enabled annotation processing.

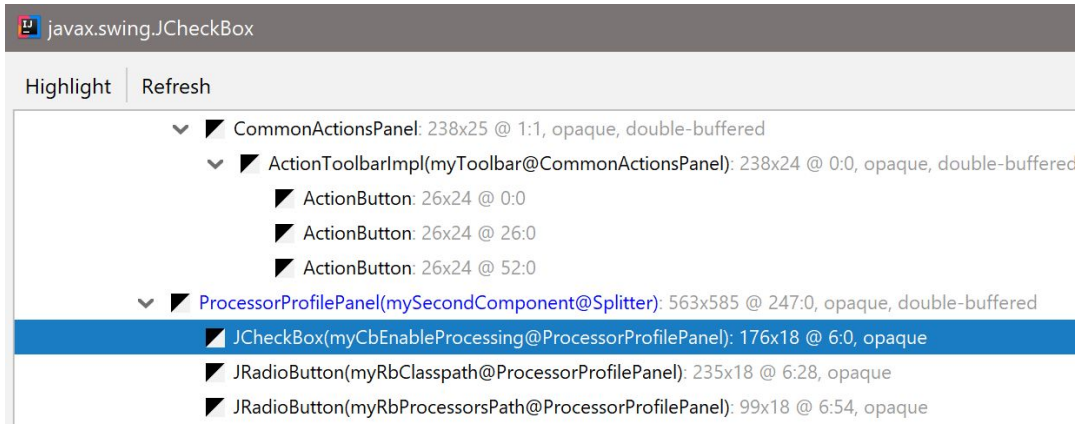
Проверим, что в Micronaut проектах включены Annotations Processors!

Находим нужную точку расширения

1. Реализуем интерфейс `com.intellij.openapi.startup.StartupActivity`:

```
<extensions defaultExtensionNs="com.intellij">
   <postStartupActivity implementation="demo.CheckAnnotationProcessorsStartupActivity"/>
```

2. Используем UI Inspector (`Ctrl+Alt+Click`):



Implicit Usages

Проблема: фреймворки нынче умные и любят неявности.

```
@Controller
class WelcomeController {
   @View("welcome")
   @Get("/")
   public HttpResponse<?> welcome() {
       return HttpResponse.ok();
   }
}
```

💡 Class ‘WelcomeController’ is never used.

Давайте научим IDE понимать неявности правильно!

Implicit Usage Provider

Регистрируем новую точку расширения:

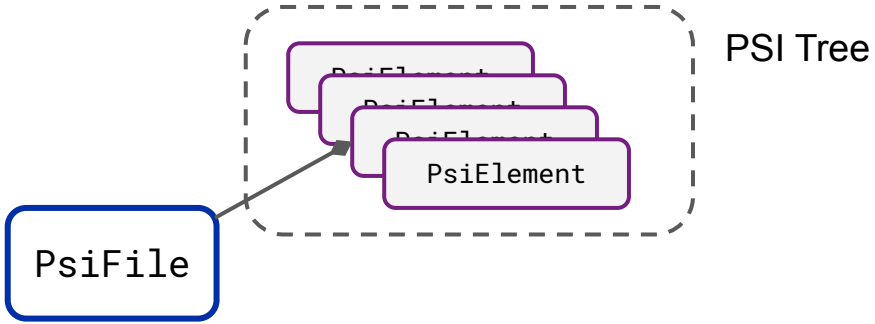
```
<implicitUsageProvider implementation="demo.MicronautImplicitUsageProvider"/>
```

```
class MicronautImplicitUsageProvider : ImplicitUsageProvider {
    override fun isImplicitWrite(element: PsiElement?): Boolean {
        TODO()
    }

    override fun isImplicitRead(element: PsiElement?): Boolean {
        TODO()
    }

    override fun isImplicitUsage(element: PsiElement?): Boolean {
        TODO()
    }
}
```

Основы работы с синтаксическими деревьями

PSI - Program Structure Interface, специальное представление:

- Синтаксические структуры проекта и кода
- Включает данные о семантике
- Всё может быть представлено в виде PsiElement (ну почти)
- Каждый язык предоставляет свои PSI элементы

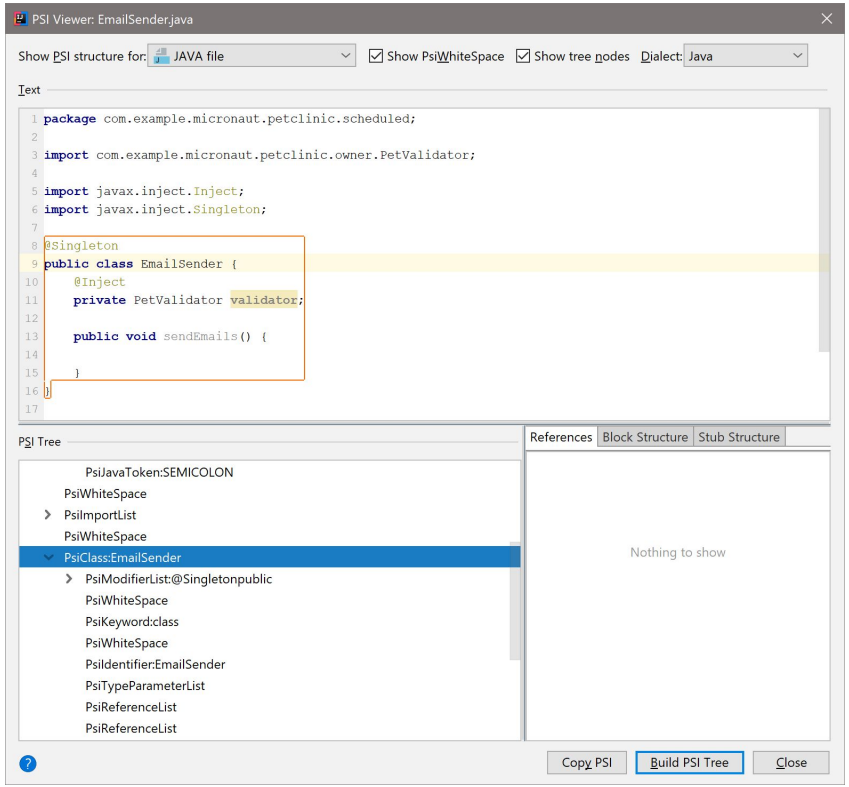


Java PSI модель

Основные элементы:

- PsiJavaFile
- PsiImportList
- PsiClass
- PsiAnnotation
- PsiField
- PsiMethod
- ...

См. Tools - View PSI Structure of Current File



Кто-то постоянно косячит в cron expressions

Пример:

```
@Singleton
public class VisitProcessor {
    @Scheduled(cron = "0 55 11 * * * .")
    public void run() {
        // do work
    }
}
```

Что тут пошло не так?

Пишем инспекции

Инспекции:

- Выполняют статический анализ кода на лету
- Обходят PSI дерево
- Могут предоставлять автоматические исправления

См.
LocalInspectionTool
LocalQuickFix

Проверим @Scheduled cron expression

Регистрируем класс инспекции:

```
<localInspection language="JAVA"
                 displayName="@Scheduled bean inspection"
                 groupName="Micronaut"
                 enabledByDefault="true"
                 implementationClass="demo.MicronautScheduledInspection"/>
```

Определяем, что хотим проверять:

```
class MicronautScheduledInspection : AbstractBaseJavaLocalInspectionTool() {
   override fun checkMethod(method: PsiMethod, manager: InspectionManager, isOnTheFly: Boolean)
        : Array<ProblemDescriptor>? {
       // check here
       return ProblemDescriptor.EMPTY_ARRAY
   }
}
```

Поддерживаем зоопарк JVM-языков

Universal Abstract Syntax Tree (UAST)

- Описывает JVM languages superset: Java, Kotlin и Groovy
- Элементы
  - *UElement, UFile, UClass, UMember, UField, UMethod, ...*
- Конструкции
  - *UComment, UDeclaration, UExpression, UBlockExpression, UCallExpression, USwitchExpression, …*
- Если не хватает возможностей - спускаемся на уровень PSI (resolve)
- Не поддерживается кодогенерация (используем PSI)

См. **org.jetbrains.uast**

UAST инспекции

1. Изменить language на **UAST** в **plugin.xml**

```
<localInspection language="UAST"
                 displayName="@Inject field inspection"
                 groupName="Micronaut"
                 implementationClass="demo.MicronautFieldInjectionInspection"/>
```

2. Получить *UElement*
3. Найти проблему в UAST дереве (или в синтетическом PSI)
4. Получить *sourcePsi*
5. Зарегистрировать проблему для элемента из *sourcePsi*

См. **Tools - Internal Actions - Dump UAST Tree**

Примеры: **intellij-community/plugins/devkit/** и **Android Studio**

Реализуем свою навигацию

Publish/Subcribe с **`@EventListener`**:

- События это сложно
- Разработчики воют
- IDE не хочет нам помогать

Идея: Иконки в gutter для навигации к publishers / subsribers!

```
import io.micronaut.runtime.event.annotation.EventListener;
import javax.inject.Singleton;

@Singleton
public class OwnerTracker {

    @EventListener
    public void onCreate(OwnerCreatedEvent event) {
        System.out.println("New owner added " + event);
    }
}
```

См.
LineMarkerProvider
RelatedItemLineMarkerProvider

Базовые индексы

- Индекс слов
  - Ключ – хешкод слова
  - Значение – битовая маска места вхождения (в коде, в комментарии, в строке)
  - Используется в Find in Path, Find Usages
- Индексы имён и типов файлов
- Индекс имён Java-классов

См. PsiSearchHelper

Как что-то нужное найти

Модуль **java-indexing-api** предоставляет методы поиска по стандартным индексам:

- `ClassReferenceSearch`
- `MethodReferenceSearch`
- `AnnotatedElementsSearch`
- …

```
Query<PsiReference> MethodReferencesSearch.search(psiMethod, scope, strict)
```

```
Query<PsiReference> AnnotatedElementsSearch.searchPsiMethods(annotationClass, scope)
```

Кэширование во спасение

Искать это **долго**! Давайте кэшировать!

```
val cacheManager = CachedValuesManager.getManager(module.project)
return cacheManager.getCachedValue(module) {
   val result = heavyFunction(module)
   CachedValueProvider.Result.create( result ,
       PsiModificationTracker.MODIFICATION_COUNT, // << track code changes everywhere
       ProjectRootManager.getInstance(module.project) // and in project structure
   )
}
```

См. CachedValuesManager

Ссылки в PSI дереве

- Позволяют установить связь между PsiElement и его использованиями:
  usage ➜ declaration
- Могут предоставлять варианты для авто дополнения
- Можно добавить ссылки без необходимости расширять язык!

```
@Controller
@RequestMapping("/owners/{ownerId}")
class PetController {

    ...

    @ModelAttribute("owner")
    public Owner findOwner(@PathVariable("ownerId") int ownerId)
```

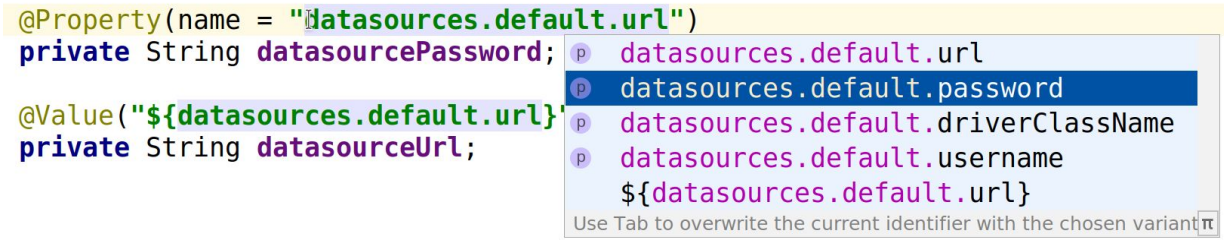
См.
PsiReferenceContributor
PsiLanguageInjectionHost

Расставляем ссылки

В Micronaut любые аннотации могут содержать подстановки свойств из **application.properties** в виде ${some.property-name}:

```
@Value("${datasources.default.url}")
private String datasourceUrl;
```

Идея: давайте свяжем такие строки со свойствами из конфига!

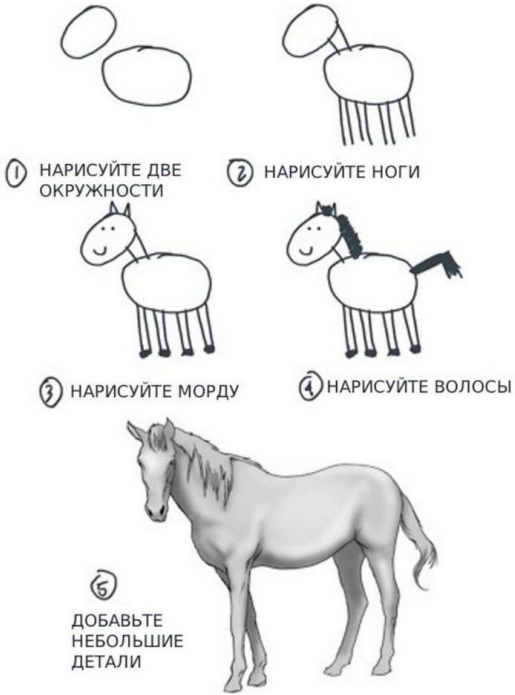


А что делать дальше?

Правильная поддержка фреймворка

Основные возможности IDE
для поддержки в плагине:

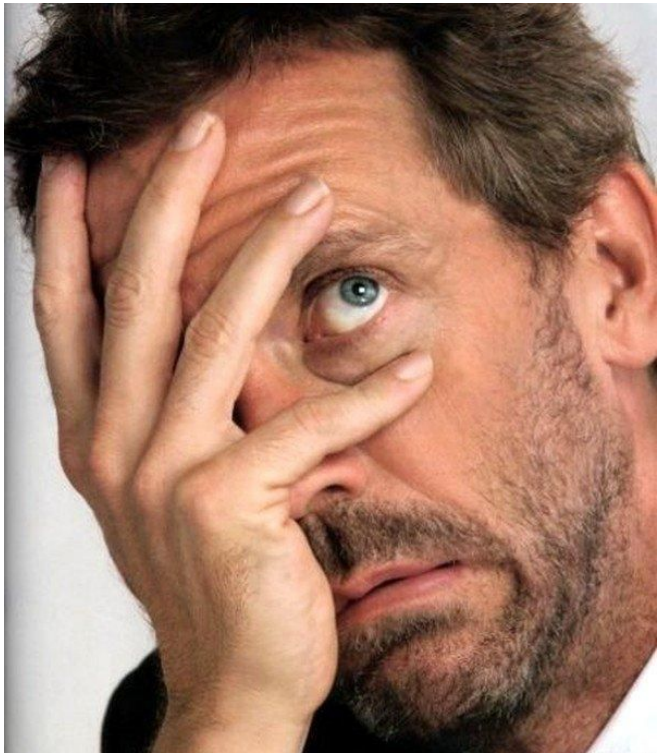
- Implicit Usages
- References
- Inspections and Quick Fixes
- Intention Actions
- Line Markers
- Language Injection
- New Project Wizard
- File Templates



Кода бывает нужно много

Spring Framework Support
в цифрах:

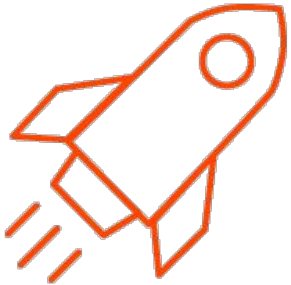
- 260k LOC Java
- 20k LOC Kotlin
- 350k LOC Total
- 3400 Tests



Распространяем плагин в компании

Достаточные условия:

- HTTP сервер (например Nginx)
- Файл `updatePlugins.xml`



```
<plugins>
   <!-- Each <plugin> element describes one plugin in the repository. -->
   <plugin id="fully.qualified.id.of.this.plugin"
           url="https://www.mycompany.com/my_repository/mypluginname.jar"
           version="major.minor.update">
       <idea-version since-build="181.3" until-build="191.*" />
   </plugin>
   <plugin>
       <!-- And so on for other plugins... -->
   </plugin>
</plugins>
```

См. Publishing a Plugin to a Custom Plugin Repository

Как можно обойтись без плагина

Если можно не писать плагин - не надо писать плагин!

А вместо этого:

- Suppress unused for class annotated with …
- Настроить инспекции и плагины
- Закоммитить директорию 📁 .idea в VCS
- Настроить Language Injection
- Использовать JetBrains annotations (nullity, contract, pure, language)

```
dependencies {
    compileOnly 'org.jetbrains:annotations:17.0.0'
}
```

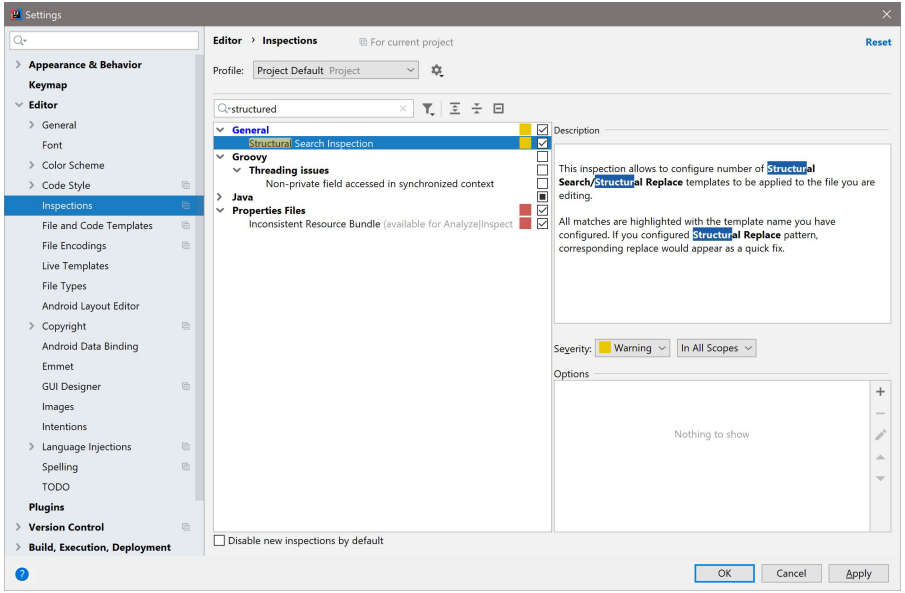
Структурный поиск для инспекций

Проверяем код без плагина:

- Включить Inspections - Structured Search Inspection
- Настроить шаблоны поиска и замены

Пример:

```
@Inject $Field$
```



Куда копать дальше

- Исходный код IntelliJ IDEA CE: github.com/JetBrains/intellij-community
- Документация Plugin DevKit: jetbrains.org/intellij/sdk/docs/basics.html
- Forum: IntelliJ IDEA Open API and Plugin Development

Вопросы ?

Исходный код:
github.com/jreznot/micronaut-intellij-plugin

Twitter:
@Yuriy_Artamonov