VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

GENERATION AND MAINTENANCE OF JAVA CLASSES BASED ON APIARY BLUEPRINT

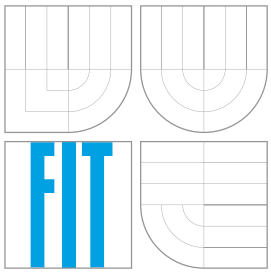
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ HERMANN

BRNO 2015

BRNO UNIVERSITY OF TECHNOLOGY
VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ



FACULTY OF INFORMATION TECHNOLOGY
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

GENERATION AND MAINTENANCE OF JAVA CLASSES BASED ON APIARY BLUEPRINT

GENEROVÁNÍ A ÚDRŽBA TŘÍD JAVA NA ZÁKLADĚ APIARY BLUEPRINTU

BACHELOR'S THESIS
BAKALÁŘSKÁ PRÁCE

AUTHOR
AUTOR PRÁCE

LUKÁŠ HERMANN

SUPERVISOR
VEDOUCÍ PRÁCE

doc. Ing. HEROUT ADAM, Ph.D.

BRNO 2015

Abstract

When developing a larger application it can be difficult to maintain code up-to-date with API changes. Sometimes the API changes without the developer being notified. Even though developer knows that the API has changed, he often doesn't know where and how. Developers who use the Apiary service for creating documentation for the API, while using a development environment from the IntelliJ IDEA family, can now download the Apiary Blueprint Manager (ABM). ABM is a plugin written in the Java language for all IntelliJ IDEA family development environments. It can accurately and in detail announce which part of the API documentation has changed and how. The plugin user is notified when for example the parameters of a request change, or when a data type of a parameter in a request changes or when a name of a parameter is changed. The plugin can also generate the necessary code for working with the API, which makes work much easier when creating data entities or requests.

Abstrakt

Při vývoji větší aplikace může být při změnách v API problém udržovat kód aktuální. Někdy se dokonce API změní, aniž by o tom byl programátor informován. A i když ví, že se API změnilo, často neví kde a jak. Vývojáři, kteří využívají služeb Apiary pro vytváření dokumentace k API a zároveň používají některé vývojové prostředí z rodiny IntelliJ IDEA, mají nyní možnost stažení Apiary Blueprint Manageru (ABM). ABM je plugin, napsaný v jazyku Java, do již uvedeného vývojového prostředí, který dokáže přesně a detailně oznámit, která část se v API dokumentaci změnila. Plugin vývojáři například oznámí změnu počtu parametrů u požadavku, změnu datového typu u parametru nebo změnu názvu parametru. Také dokáže generovat kód potřebný pro práci s API, což značně ulehčí práci při vytváření datových entit nebo požadavku.

Keywords

ABM, Apiary Blueprint Manager, API Blueprint, Keep code up-to-date, API analysis, Plugin development

Klíčová slova

ABM, Apiary Blueprint Manager, API Blueprint, Udržování aktuálního kódu, Analýza API, Vývoj pluginu

Citace

Lukáš Hermann: Generation and Maintenance of Java Classes Based on Apiary Blueprint, bakalářská práce, Brno, FIT VUT v Brně, 2015

Generation and Maintenance of Java Classes Based on Apiary Blueprint

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana doc. Ing. Adama Herouta, Ph.D..

.....
Lukáš Hermann
May 13, 2015

Poděkování

Chci poděkovat především svému vedoucímu práce, který pro mne byl v průběhu tvorby oporou.

Contents

Chapter 1

Introduction

One of the most common problems that application developers face is the access to data outside the application. These issues are usually handled by using an API. Web services are usually accessed using REST API based on the HTTP. REST is a design-pattern for applications exposing their API via HTTP. REST API being stateless allows for parallel request processing. It defines methods to perform CRUD operations on the data exposed via API.

It is a good practice to have an accurate documentation of all web API calls before starting implementing those requests in the application code. Apiary is one of such services that provides a possibility of generating documentation based on API Blueprint, which is language based on the Markdown language, used for describing web API. Based on this, or any other documentation it should no longer be a problem for developers to connect applications with a Web service to process, or to obtain data from it.

Let's assume that there are more developers working on a project. One of them is a web API developer, and the second one is in charge of implementing the client application. There is the problem of keeping code up-to-date with the web API. Developer of the web API can change the interface at any time. If he won't let know the client application developer about it as soon as possible, it could be fatal for the client application, and it could start crashing on some API calls that have changed.

This example scenario illustrates the problem that has to be solved. It is key to ensure that the application developer is always well and in time informed about web API call changes.

1.1 Motivation

Nowadays, there is no easy solution for an application developer to check whether there has been any change of web API he is using in his application. Even if he knew there is a change, he would not know where exactly the change is, how and where it has affected his code, and what he has to change to make it work again. Solution for such situations will help a lot of developers to keep their code up-to-date with the API, limit application crashes, and save a lot of time spent by checking documentations, and looking for what has changed since the last time.

1.2 Structure of the Document

Chapter 2 generally describes API, REST API, what is Apiary, and what Apiary has to do with API. There is also a brief introduction to API Blueprint.

Brief introduction to what are Integrated Development Environments, IntelliJ IDEA, Android Studio, and how to create a plugin for these IDEs will be presented in chapter 3.

Chapter 4 deals closely with the problem this thesis is solving, and describes the solution of keeping code up-to-date with web API by creating a plugin for an IDE. This chapter includes problem description, solution design and description of the implementation of each part of the plugin.

Testing disclosed in chapter 5, shows the impact of usage of this plugin on work efficiency.

The last chapter 6, contains conclusion of this thesis.

Chapter 2

API, REST API, Apiary and API Blueprint

API (Application Programming Interface) is a term used in information technology for collection of methods, classes and protocols of a unit (for example a library, or a service) a developer can use. That means that API tells us how to communicate with that unit, without knowing its exact implementation. It is an interface between two separate pieces of software. In this thesis we are dealing with web API, more precisely about HTTP based RESTful APIs.

2.1 REST API

Roy Thomas Fielding stated in his dissertation thesis [1] that REST (Representational State Transfer) is an architectural style for distributed hypermedia systems, describing the software engineering principles guiding REST and the interaction constraints chosen to retain those principles, while contrasting them to the constraints of other architectural styles.

RESTful APIs are APIs, which adhere to REST architectural constraints. Those constraints are:

- Client-Server (Client request resources or service from Server).
- Stateless (Threat each request as independent transaction).
- Cache (Can eliminate some of the Client-Server requests).
- Uniform Interface (Simplifies and decouples the architecture).
- Layered System (A client cannot tell whether it communicate with the end server, or with an intermediary server).
- Code-On-Demand (Server can send client an executable code).

By Roy Thomas Fielding blog post [2], the HTTP based RESTful API:

- should not be dependent on any single communication protocol and should not contain any changes to the communication protocols aside from filling-out or fixing the details of underspecified bits of standard protocols.
- should spend almost all of its descriptive effort in defining the media type(s) used for representing resources and driving application state, or in defining extended relation names and/or hypertext-enabled mark-up for existing standard media types.
- must not define fixed resource names or hierarchies.
- should never have "typed" resources that are significant to the client.
- should be entered with no prior knowledge beyond the initial URI (bookmark) and set of standardized media types that are appropriate for the intended audience.

For example, the API call to obtain weather data can look like this:

```
GET /weather/location?country=cz&city=prague
Host: http://someweatherpage.com
```

In case server returning JSON data, the response could look like this:

```
{
   country: "cz",
   city: "prague",
   weather: "sunny",
   temperature_celsius: 24,
   temperature_fahrenheit: 75.2
}
```

2.2 Apiary

Apiary is a company that provides a platform for the design of API life-cycle. It was founded in 2010 in the Czech Republic, but now resides in San Francisco, USA. Apiary is creator of API Blueprint. [3]

2.3 API Blueprint

Based on the documentation on GitHub [4], API Blueprint use a simple markdown syntax based language that is enhanced by GitHub Flavored Markdown syntax. This language is then used for describing the web API.

To better understand the API Blueprint, here is a basic example of the API Blueprint:

```
FORMAT: 1A
HOST: http://some.host.com

# Mock API
This is a example blueprint for my thesis
```

At the beginning of each API Blueprint is a metadata section. For this API Blueprint example `FORMAT` was set to value `1A`. `FORMAT` keyword denotes the version of the API Blueprint. There is also a `HOST` definition specifying URL of targeted API.

The first heading in the API Blueprint defines the name of the API. Number of `#` symbols defines the level of the heading. In this example the name is "Mock API". Heading can be followed with a description on a new line.

```
# Group Notes

## Note [/note]
Create new note.

+ Model (application/json)

    JSON representation of Note.

    + Body

            {
                "id" : 666,
                "name" : "Test note",
                "text" : "Some text inside note"
            }
```

The `Group` keyword at the start of the heading allows for a section for related resources to be created. In this case, `Notes` group contains documentation of the resources working with notes.

To define a resource, one needs to provide a resource name and URI used to access this resource, placed inside square brackets at the end of the heading. Resource can have a description placed on a new line right after the heading as well.

It is also possible to create a model, which can be later used to represent some data. This example illustrates a model for the Note entity that has `\application/json"` Content-Type, and JSON body containing three elements.

```
### createNote [POST]
Create new Note

+ Request

    [Note][]

+ Response 201
```

The `createNote` heading is an action in `Note` resource that uses POST HTTP method. As input, it takes a `Note` entity, specified in the `Note` model. Possible response from this action is a HTTP status code `201`.

```
1
2  ## Note [/note/{id}]
3  Get old note by ID.
4
5
6  ### getNote [GET]
7  Get note by ID
8
9  + Parameters
10     + id (integer) ... ID of note to retrieve.
11
12 + Response 200
13
14     [Note][]
```

The last example is a resource that belongs to the `Notes` group. This example demonstrates the usage of parameters. They need to be specified in the resource's URI address, and then defined in Action as Parameters.

Chapter 3

IDE and Plugin Development for InteliJ IDEA

IDE (Integrated development environment) is single program which provides many features for modifying, compiling, debugging and deploying software. IDE also usually provide many separate tools, such as GCC, make, or ant. If developer uses the right IDE, he won't need any other tool to complete his designated task.

IDEs can be divided into single-language (in case they are dedicated to a specific programming language, e.g. Xcode and Delphi) and multi-language (in case they support multiple languages, e.g. InteliJ IDEA, Eclipse, NetBeans and Microsoft Visual Studio) IDEs. They can also be divided into graphical and text-based IDEs.

3.1 IntelliJ IDEA

Information accessible on official JetBrains site [6] says that IntelliJ IDEA is a Java IDE, that claims to be the most intelligent Java IDE. There are free (Community Edition) and paid (Ultimate Edition) versions available for download. IntelliJ IDEA provides:

- Productivity-Boosting Features (Smart Code Completion, On-the-fly Code Analysis, Advanced Refactorings)
- Developer Tools (Database Tools[1], UML Designer[1], Version Control Tools, Build Tools)
- Web Development (offers advanced support for the most important web frameworks and standards)[1]
- Enterprise Development (offers an out-of-the-box tool set for building enterprise applications)[1]
- Mobile Development (Android, AIR Mobile[1])
- Agile Development[1]

[1] Available only in Ultimate Edition.

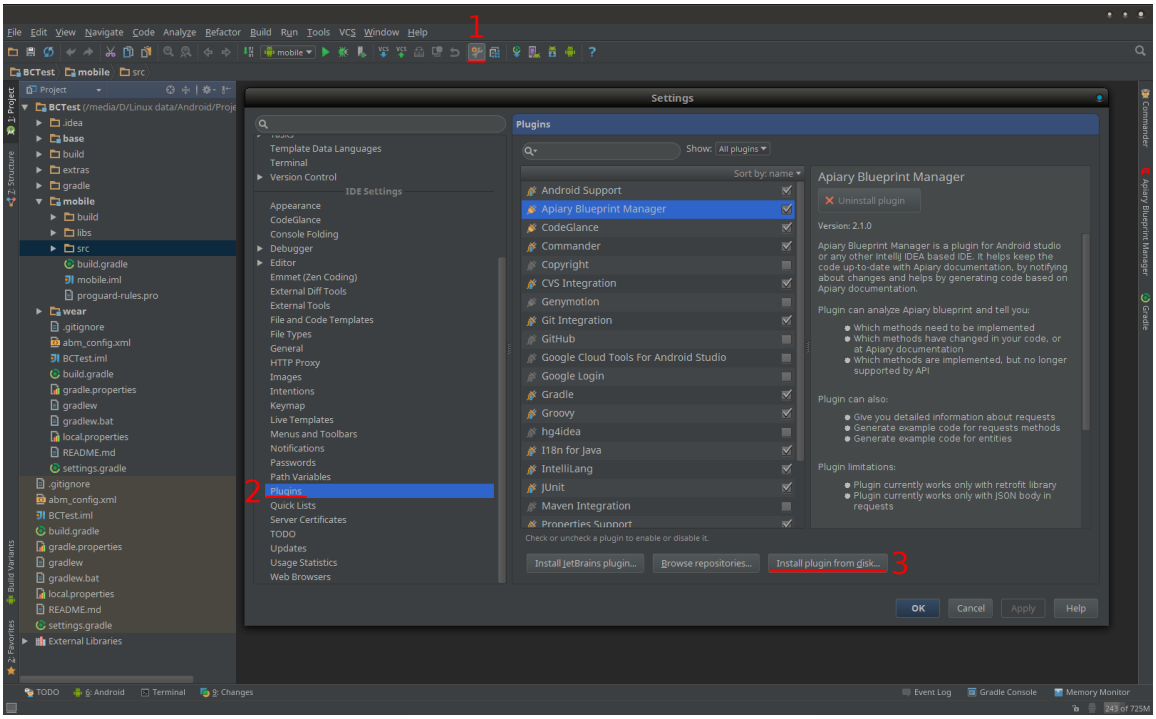


Figure 3.1: Android studio with three easy Plugin installation steps.

3.2 Android Studio

Based on an official Android Studio page [5], Android Studio is the official IDE for Android application development, based on IntelliJ IDEA, developed by Google. Android Studio replaced Eclipse Android Development Tools (ADT) as a primary IDE for Android application development. Android Studio offers for example these features:

- Gradle-based build system
- WYSIWYG Layout editor
- ProGuard and app-signing capabilities
- Code templates to build common app features
- Support for Android Wear development

3.3 Plugins and Plugin Development for InteliJ IDEA

Plugin (can be also called Extension or Add-on) is a piece of software that adds a specific feature to an existing software application. A lot of applications support plugins. They can be found in web browsers, IDEs, Office Suites, media players, etc. They don't necessarily add new features. They can just change the way an application looks.

In case of plugins for IDEs, they usually add support for generating code, code refactoring, code inspection, building and testing.

All IntelliJ IDEA based IDEs support the same plugin type. So plugin made in IntelliJ IDEA can also be installed in Android Studio, RubyMine, PhpStorm, PyCharm, etc. Plugin is developed in IntelliJ IDEA, and the output file has the .jar extension.

Before starting developing a plugin, there are some requirements that need to be met:

- IntelliJ IDEA IDE version 9.0 or later
- IntelliJ IDEA SDK have to be configured for IDEA project
- "Plugin DevKit" plugin enabled in **Settings** → **Plugins**

Before creating a new project, the IDE needs to be configured properly. This configuration can be done in Project Structure dialog. These two things need to be configured:

- One has to add new "IntelliJ IDEA Plugin SDK" in SDKs menu item, with correct path of IntelliJ IDEA installation directory.
- Also, sources of IntelliJ IDEA Community Edition must be downloaded, and path to it must be set in "Sourcepath" tab.

After configuring IntelliJ IDEA properly, the IDE is ready for creating a new plugin project. This is done by selecting **File** → **New Project**. In the project creation dialogue the IntelliJ Platform Plugin must be selected as well as the correct Project SDK has to be chosen.

Creating the project this way will automatically create a plugin.xml file in the META-INF folder, located in the root directory of the currently used project module. This file contains information about the plugin like name, version, vendor and plugin description. There are also sections for registering actions, project and application components.

Plugin can be easily tested by selecting **Run** → **Run Plugin**, which opens new IntelliJ IDEA IDE instance where the plugin is already loaded. Plugin can be debugged the same way by selecting **Run** → **Debug Plugin**.

In order to deploy a finished plugin, one must build the final JAR file. This is done by selecting **Build** → **Prepare Plugin Module** <**name**> **For Deployment**. This generates the JAR file and shows the path to where it is stored (usually in module root directory). This file can now be installed in any IntelliJ IDEA based IDE, by selecting **File** → **Settings** → **Plugins** → **Install plugin from disk**.

The result of your hard work can be also shared with other developers over **JetBrains**[2] repository. In order to be allowed to upload your own plugin you need to create a free account on JetBrains' site. After logging in, you simply upload your JAR file to be shared with others.

[2]JetBrains repository can be found at: https://plugins.jetbrains.com/

Chapter 4

Problem and Solution Using Plugin

4.1 Problem

The main problem this thesis solves is how to keep the application code up-to-date with API specification. How to warn the client application developer that something has changed in the API. A second problem to solve is how to help the application developer with implementation of the basic actions the API allows for in his code.

4.2 Suggestion of Solution

Before the implementation of a plugin, there are a few things that need to be realized. The first thing to become conscious of is that in prior to managing an API Blueprint, you need to get one. After obtaining it, it has to be translated into something that can be easily converted into a entity structure. Plugin has to take this representation of data, and compare it to the code in the project. After comparing the data, plugin has to show what has changed, what is not implemented and what was implemented, but is no longer needed. Also, plugin should be able to construct code for entities and requests handling based on those data, in order to save maximum of the user's time. As for user interface design, it is wise to implement this type of plugin as a `ToolWindow` class (4.3.4).

API Blueprint can be obtained from the official Apiary site, by providing a correct pair of API name and token that can be generated on their site. But API Blueprint doesn't have to be always on Apiary site. It can be produced in any text editor. So adding support for local file sources, and URL address for text file is a must.

API Blueprint obtained from Apiary site, or from URL address is obtained via network. For this the ABM plugin uses the Unirest framework which is very easy to use, and can make HTTP requests with methods GET and POST, which are used for obtaining and parsing the API Blueprint.

After obtaining the API Blueprint, the plugin needs to process it into a more suitable form in order to work with the data efficiently. JSON format is great for this. A parser service is hosted on `https://apiblueprint.org` that takes API Blueprint as input, and yields JSON data as output. The plugin leverages the Unirest framework once again for sending data to the parser, and receiving the response.

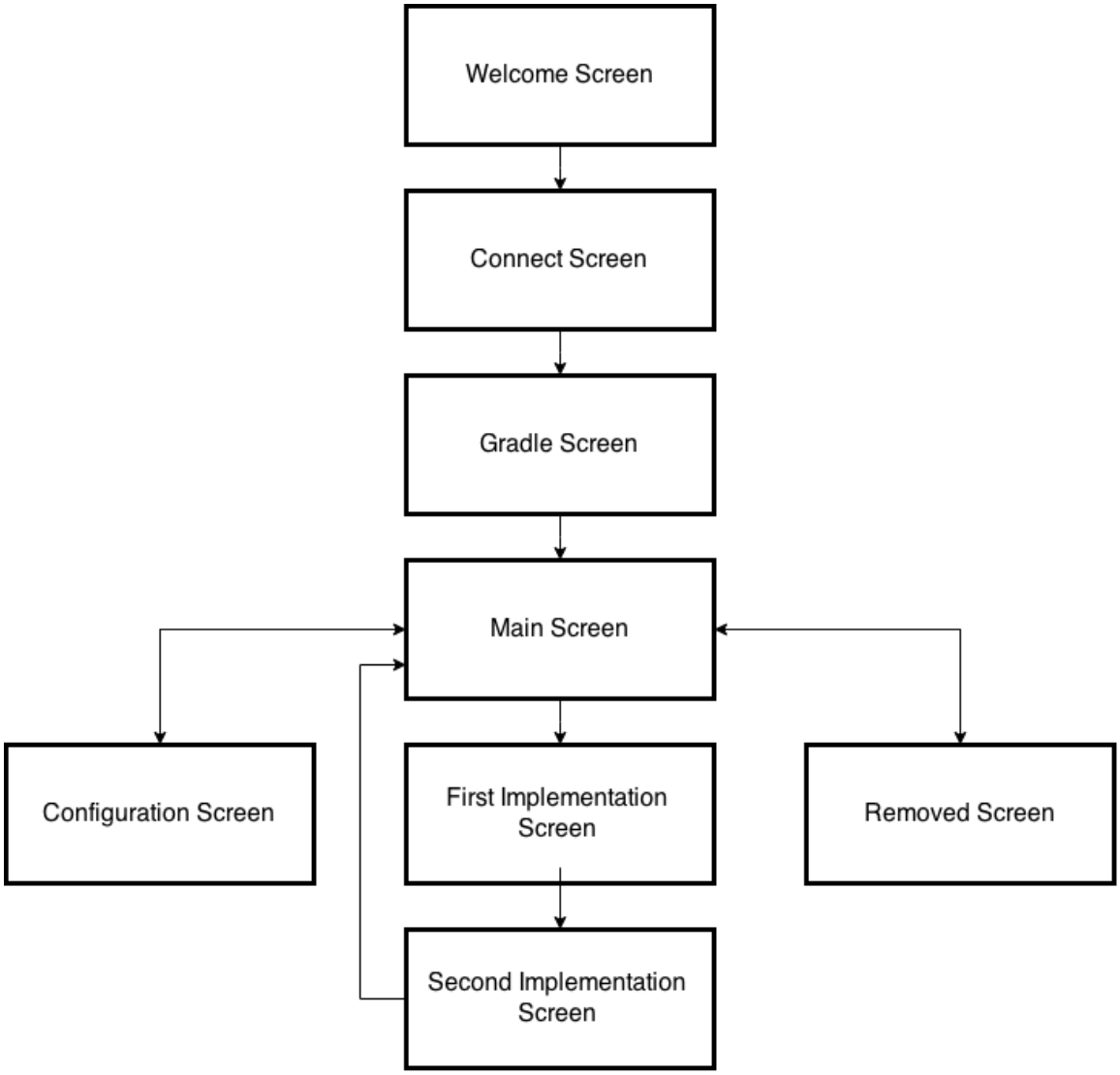


Figure 4.1: Flow diagram of all plugin screens.

A method must be implemented for analyzing the API Blueprint, this method has to go through the data entities parsed from the API Blueprint and compare them to the methods and entities in the project code. For that the plugin uses methods from `ProjectManager` class 4.3.3. This class is located in plugin Utility package, and contains utility methods for manipulating with project files, and contains methods like `checkMethodForProblems` and `checkEntityForProblems`.

All this is going to be implemented using PSI (Project Structure Interface) classes. These PSI classes provide easy verification of the project code. By using this, the plugin is able to check whether the package contains a specific class, and verify methods in this class, their parameters, etc..

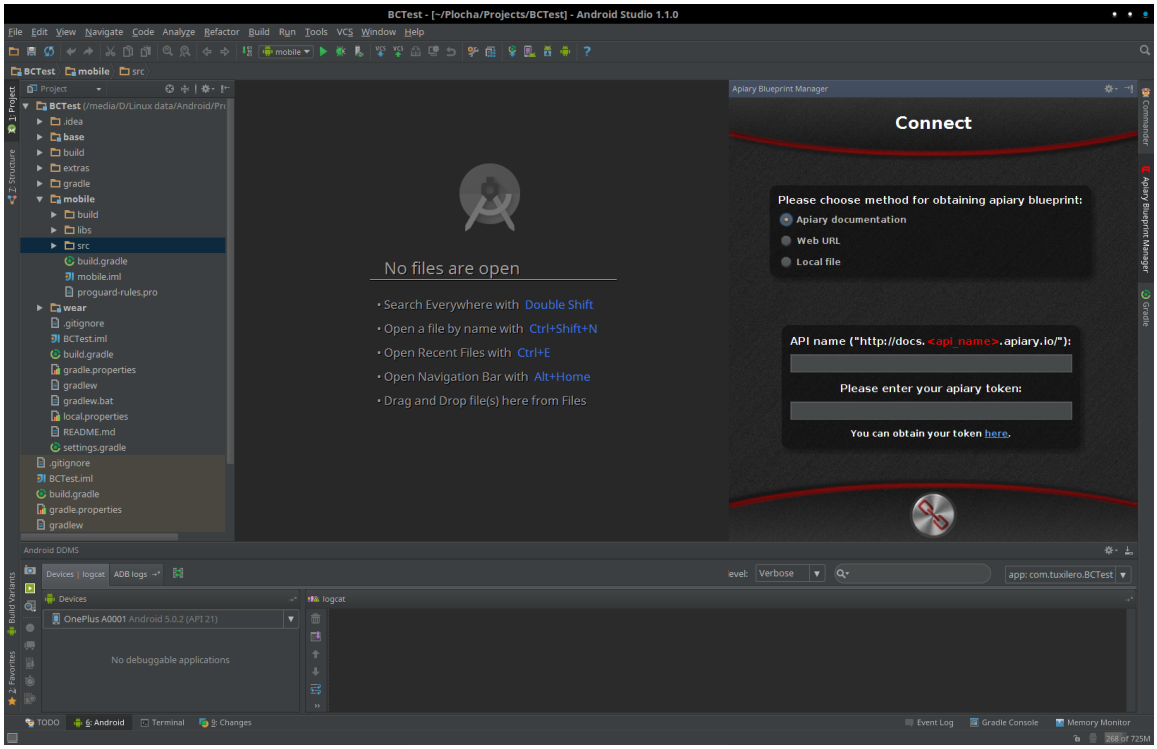


Figure 4.2: ABM Plugin with Connect screen (Figure 4.4) opened in Android Studio, showing the color tuning of Plugin with Android Studio.

From this analysis the plugin will generate a list of items indicating the state of the code. Clicking an item will cause the plugin to offer a solution to the problem. In the implementation screen the developer has to fill in manually the entities? name because there is no way of generating it automatically. With this information provided the plugin can generate example code for the request method, and all the entities needed for this request.

For the plugin in order to be able to recognize what has changed, and to generate appropriate code, it needs to have an exact syntax. Because of this the plugin is limited for usage only with the Retrofit framework.

It is important for a plugin that is used often to provide a great experience. Hence, it was important to choose colors that fit into Android Studio or IntelliJ IDEA. As most people use a dark theme in their IDE, I've decided to choose dark grey colors, combined with red elements and white fonts which go perfectly with a dark theme as can be seen on (Figure 4.2). Black boxes that can be seen in the image (Figure 4.2) were created using 9-patch images that can be stretched to fit in content. For the button two images were used: one lighter and one darker for the pressed state.

Also, some buttons indicate progress, like rotating connect button when connecting, or rotating refresh button on the Main screen (Figure 4.6), which indicates that API Blueprint is being downloaded and refreshed. GIF images were used for such animations.

The whole UI was made using Swing, which is a GUI widget toolkit for Java. UI was laid out using MigLayout, which is Java Layout Manager for Swing. One of the challenges in laying out the UI was to fit the plugin content size to the resolution of a screen. Using fixed sizes of font and blank spaces resulted in huge buttons, text, etc. on screens with lower resolutions. This is solved by utility methods for computing font size and dimensions based on the current resolution so that the plugin always looks adequate for the used screen resolution.

Another problem was to make background repeat, and to have the 9-patch images in correct sizes. 9-Patch image is stretchable, repeatable image which contains a 1px border. The colors in the border determines if a piece is static, stretches or repeats. To make this work the class `JBackgroundPanel` extends `JPanel` and overrides `paintComponent` method, and renders background correctly based on type, specified in the constructor.

Last problem in the UI design were the image buttons. The challenge is to scale them properly to fit in the layout. Animated buttons using GIFs as source proved to be the biggest challenge in the user interface rendering – particularly the resizing of such buttons. It is solved in the class `ImageButton` that extends `JButton`. The only way to make button to have an animation is to provide it with an icon. The GIF needs to be resized frame by frame in order for this to work. If treated the same way a static image would be treated, it would have lost its animation.

4.3 Implementation

This section describes in detail the project structure, and the implementation process of this plugin, including solution of problems encountered during development of this plugin.

4.3.1 Project Structure

A typical Java project has the following structure:

- `src/main/java/` – containing packages (directories) with source files.
- `src/main/resources/` – containing packages with resource files, like images, xml files, property files. These files are data resources which can be easily accessed by the application code.
- `libs/` – directory containing libraries that are used by the project.

Plugin project sources are sorted into a few packages:

- Entity – contains all entity classes for the plugin project. It also contains two subpackages with entities for API Blueprint and configuration.
- Enums – contains all the enums used in the plugin project.
- Renderer – contains `ABMTreeCellRenderer`, which is used to render tree node item in main window.
- UI – contains all classes rendering UI.
- Utility – contains utility classes:
  - `ConfigPreferences` – class that manages operations with configuration file (4.3.2).
  - `Log` – class that helps with printing debug information.
  - `Network` – class for network communication.
  - `Preferences` – class managing saving and loading preference values.
  - `ProjectManager` – class containing many useful methods for the plugin implementation (like computing sizes for fonts, providing access to resource files, file access, generating messages, etc.) (4.3.3).
- View – contains custom views `ImageButton` and `JBackgroundPanel`.

There is the `ABMConfig` class in root package, which contains constants like plugin version, temp file names, and switches for debugging and logging.

Plugin project resources are sorted into these packages:

- Drawables – all images used in plugin project.
- META-INF – plugin.xml file with plugin configuration and information about the plugin.
- Values – color and string property files, which contains colors and texts used in plugin project.

4.3.2 Plugin Configuration

Plugin needs to store information about configuration, and already implemented requests somehow. `ConfigPreferences` class was implemented to cater for that. It contains methods for saving and loading configuration file abm_config.xml. The configuration file is saved in project root directory as an XML file, so that it's easy to read even for the plugin user, in case a developer would need to edit it manually.

The configuration file stores following information:

- Configuration – Entity package, Host URL, Interface class name and Project module.
- Implemented class info list – Information about each implemented requests. Whether request is asynchronous or hidden, request method, name, URI, Request and Response lists containing serializable and user defined entity names.

There are also methods `tryToFillTreeNodeEntity` and `getAllConfigEntities` for filling `TreeNodeEntity` from configuration file, and `saveTreeNodeEntity` for saving `TreeNodeEntity` into configuration file.

4.3.3 ProjectManager Class

`ProjectManager` class implements methods for managing, and checking user project files. It also contains a lot of utility methods for listing Modules, Packages, Directories and Classes.

Most interesting methods in this class are `checkMethodForProblems` and `checkEntityForProblems` which scan respectively methods and entities for inconsistencies between user code and API Blueprint. Both methods start by creating `ProblemEntity` list. Each `ProblemEntity` contains problem name and description. Once a problem is detected, it is added to this list, which is then returned as an output of those two methods.

Method `checkMethodForProblems` takes `TreeNodeEntity` as a parameter and has the following logic:

1. Check whether method with provided name exists in the Interface class specified in the Configuration screen (Figure 4.7). If the method doesn't exist, method creates and returns new `ProblemEntity` informing about this.
2. Compare method headers with headers specified in API Blueprint.
3. Compare the method?s HTTP method (GET, POST, eg.) with the method specified in the API Blueprint.
4. Check method return type.
5. Get `PsiParameterList` of an implemented method, `ParametersEntity` list from `TreeNodeEntity` and `BodyObjectEntity` list from `TreeNodeEntity`:
   (a) Loop through all `PsiParameterList` items of the implemented method.
   (b) If current item matches an item from `ParametersEntity` list or `BodyObjectEntity` list taken from `TreeNodeEntity`, remove it from the `PsiParameterList`. Otherwise, add it to problem list.
6. At the end loop through `ParametersEntity` list and `BodyObjectEntity` list, and add every item left in these lists as not implemented parameters respectively body items.

Method `checkEntityForProblems` has the following logic:

1. Check whether entity with the same name exists in entity package specified in Configuration screen (Figure 4.7).
2. Loop trough all entity variables.
3. Check whether the variable exists in `BodyObjectEntity` list of `TreeNodeEntity`.
4. If it exists, check the variable data type and annotation, and remove it from `BodyVariableEntity` list.
5. At the end loop trough BodyVariableEntity list, and add all that's left as not implemented to the problem list.

4.3.4 Plugin Initialization

First thing one must do when creating a plugin is to register an extension in the plugin.xml file. This extension is in our case registered as a <`toolWindow`> element. This means that plugin will appear in tool section of IDE, and will behave as a tool window as can be seen on (Figure 4.2). This element has the attribute `factoryClass` which accepts a class as value. This class is initialized immediately after clicking the plugin button in the IDE. In the plugin there is the class `ABMToolWindow` which implements `ToolWindowFactory`. This class has a simple logic that checks whether plugin configuration file exists using the abovementioned `ConfigPreferences` class (4.3.2) which is taking care of saving and loading data in abm_config.xml file, and based on this fact call either `ABMToolWindowMain` (4.3.8) or `ABMToolWindowWelcome` (4.3.5) constructor.

Figure 4.3: Welcome screen with some links and changelog displayed.

4.3.5 Welcome Screen

Each screen is initialized by the `initLayout` method in which the whole screen is rendered. It is the same for every screen, the logic stays the same, only difference is the content of the `middlePanel`, which is a `JPanel` located on every screen, containing the UI elements between the top and the bottom panel. First, the `JBackgroundPanel` class is created with background image and with appropriate `BACKGROUND_REPEAT` type. This UI element is then set as the `ToolWindow` instance content. The `ToolWindow` instance has to be transmitted from screen to screen. This instance of `JBackgroundPanel` than has MigLayout set as the layout manager.

Layout consists of three rows and one column. Top and bottom elements also contain `JBackgroundPanel` with appropriate background image for top and bottom box. Top panel then contains header text, and bottom panel usually contains one or two buttons, depending on the displayed screen.

In case of the Welcome screen (Figure 4.3), the `middlePanel` contains just few `JLabels` for displaying text. The problem that needed to be solved here, is how to place a clickable URL link into a `JLabel`. That is solved by replacing `JLabel` with `JEditorPane` which supports HTML tags, and by adding a `HyperLinkListener`. Plugin is then able to recognize that a link was clicked, and display the selected URL in a default web browser. This method is also used in other screens that need to have a clickable hypertext link in text.

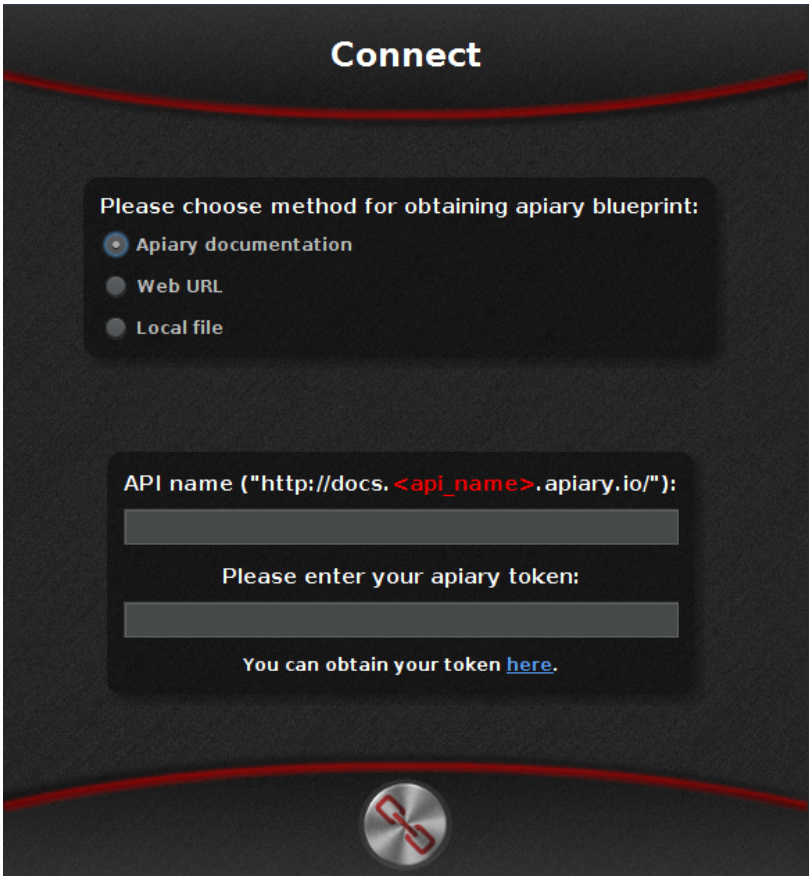


Figure 4.4: Connect screen with three `JRadioButtons` and `JTextFields`.

4.3.6 Connect Screen

As mentioned above, the layout and logic of every screen is the same. So the difference here is that now the `middlePanel` contains black boxes that are created by instantiating a `JBackgroudPanel`, with a 9-patch background image, and its type set to `NINE_PATCH`.

There are three radio buttons that are used for for selecting a connection method. After switching to another method, the bottom box changes based on which radio button was selected. This is implemented by creating `JPanel` using CardLayout as layout manager. There are three `JBackgroundPanels` that use MigLayout as usual that are added to this CardLayout, and then they can be easily switched by using the show method of the CardLayout.

After clicking the connect button, plugin starts to process provided data based on which connection method is selected. The logic is very similar in all three cases:

1. Obtain API Blueprint from selected source. In case of Apiary documentation, plugin uses `Network` class method `requestBlueprintFromApiary`, in case of Web URL plugin uses `Network` class method `requestBlueprintFromURL`, else plugin use Utils class method `readFileAsString`.

2. After obtaining the API Blueprint, its validity has to be verified. For this plugin calls `Network` class method `isBlueprintValid`.

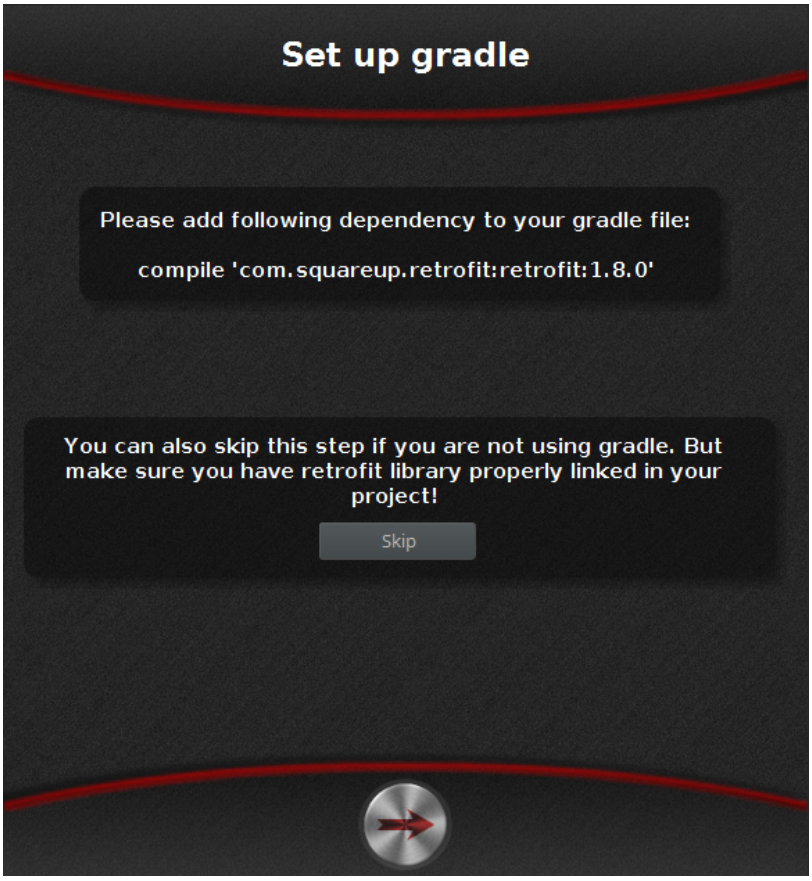


Figure 4.5: Gradle screen with information and skip button displayed.

3. In case everything is OK, API Blueprint is valid, plugin saves API Blueprint into a temp file using Utils class method `saveStringToTmpFile`, and sets Preferences values for temp file location, connection type, path and in case of Apiary documentation connection type it also saves Apiary token.

4. After saving and processing all information, plugin proceeds to the Gradle screen (Figure 4.5).

4.3.7 Gradle Screen

This screen is really simple. All it does is informing the developer that he has to use Retrofit framework in his project in order for it to work correctly. There is also a button for skipping this step, so that the plugin won't check for Retrofit framework in gradle.build file because not everyone uses Gradle in their project. Gradle is powerful build tool, based on the Groovy programming language.

When next button at the bottom of the screen is clicked, plugin checks gradle.build file for Retrofit framework occurrence using Utility class method `isGradleWithRetrofit`. In case it is missing, plugin displays a dialog informing the user about it, otherwise plugin shows the Main screen (Figure 4.6).

Figure 4.6: Main screen showing all possible `JTree` nodes except "Configuration Problems" node.

4.3.8 Main Screen

Main screen (Figure 4.6) is the most complicated of all screens. The main logic of the plugin is implemented in this screen. As for layout, there is a change in top panel, where two buttons are rendered with an image in the center, instead of a `Jlabel` header. Clicking the left-hand-side button will pop up a dialog with information about the plugin, and clicking right-hand-side button would display plugin configuration. The central image is showing the state of the code. There are three different image types:

- Error (Cross) – this image is displayed only if API Blueprint is not valid, or if there are problems in the plugin configuration.
- Warning (Exclamation mark) – this image is displayed if there are inconsistencies between the code and the API Blueprint.
- OK (Tick) – this image is displayed in case everything is ok, and when there is nothing to display, or only requests that user has marked as hidden.

`middlePanel` on this screen contains `JTree` class for displaying errors and inconsistencies between the code and the API Blueprint. In order for the text to be rendered as colored and different sized, the class `ABMTreeCellRenderer` has been implemented in which text is resized and colored based on `TreeNodeEntity` type. Data for this `JTree` are generated using `initTreeStructure` method which constructs tree items based on `TreeNodeEntity` list.

`TreeNodeEntity` list is a list of entities which stores all values for each request parsed from the API Blueprint. It also stores some values for displaying in the tree, like the text that is displayed in the tree (For example: "`Method: GET URI: /note`"), or information whether the node is visible or not.

The logic for loading the API Blueprint, parsing and displaying the code state is following:

1. Load API Blueprint from the temp file in case of just displaying, or download it again in case of reloading using the reload button. This means that if user just changes the screens, and returns to the Main screen (Figure 4.6), it won't download API Blueprint again, and the old, stored one will be used. To check and download API Blueprint again, one has to click the refresh button at the bottom of the screen.

2. Immediately after loading the API Blueprint, it has to be converted to the JSON format which is done by calling the `Network` class method `requestJSONFromBlueprint`. This method takes the API Blueprint text, and sends it to an official API Blueprint parser at `http://api.apiblueprint.org/parser`. This parser then returns a JSON representation of the API Blueprint.

3. Having this JSON representation the plugin needs to parse it into entities to be able to work with it. This is done by using Utils class method `parseJsonBlueprint` which returns `ABMEntity` object, parsing it using GSON library.

4. Having this object plugin calls the `analyzeBlueprint` method, which takes the `ABMEntity`, and analyzes it, converting it to a list of `TreeNodeEntity` instances.

5. This `TreeNodeEntity` list has to be analyzed before it can be displayed in the `JTree`. For this plugin calls the `analyzeTreeNodeList` method.

6. The last thing to do with this list is to display it into the `JTree` by creating a new `JTree` with new values returned from the `initTreeStructure` method.

The `analyzeBlueprint` method converts an `ABMEntity` instance to a `TreeNodeEntity` instance by taking only the information the plugin needs, and throwing everything else away.It also parses the Request and Response example JSON data, to `BodyObjectEntity` because the plugin needs this data later to be able to generate example entities to save the Request and Response data. This parsing is done by using GSON library, by parsing data to an Object instance where Object is the root of the class hierarchy. Parsing JSON data this way, allows the plugin to check which instance is each value, and recursively parse this object into `BodyObjectEntity` with appropriate data types and names.

`analyzeTreeNodeList` is method which compares the `TreeNodeEntity` list with the current project code state. First, this method checks using `ProjectManager` class whether entity package and interface classes are configured properly. If not, the output of this method will be the `TreeNodeEntity` list containing one, or two items in case both are configured wrong. This item will contain an error text informing that entity package and/or interface class is not configured properly. Then the method sets the correct `TreeNodeType` to the `TreeNodeEntity` instance, so that it is displayed correctly in `JTree`. This is done by using `ProjectManager` class, which can tell whether there are inconsistencies between the local project code and the API Blueprint.

Selecting an item in the `JTree` runs the following action depending on type of `TreeNodeEntity`:

- Configuration problem node will brings up the Configuration screen (Figure 4.7).
- Not implemented node and modified node show the First Implementation screen (Figure 4.8).
- Removed node shows the Removed screen (Figure 4.10).
- Hidden node shows a dialog for removing `TreeNodeEntity` from the hidden nodes.

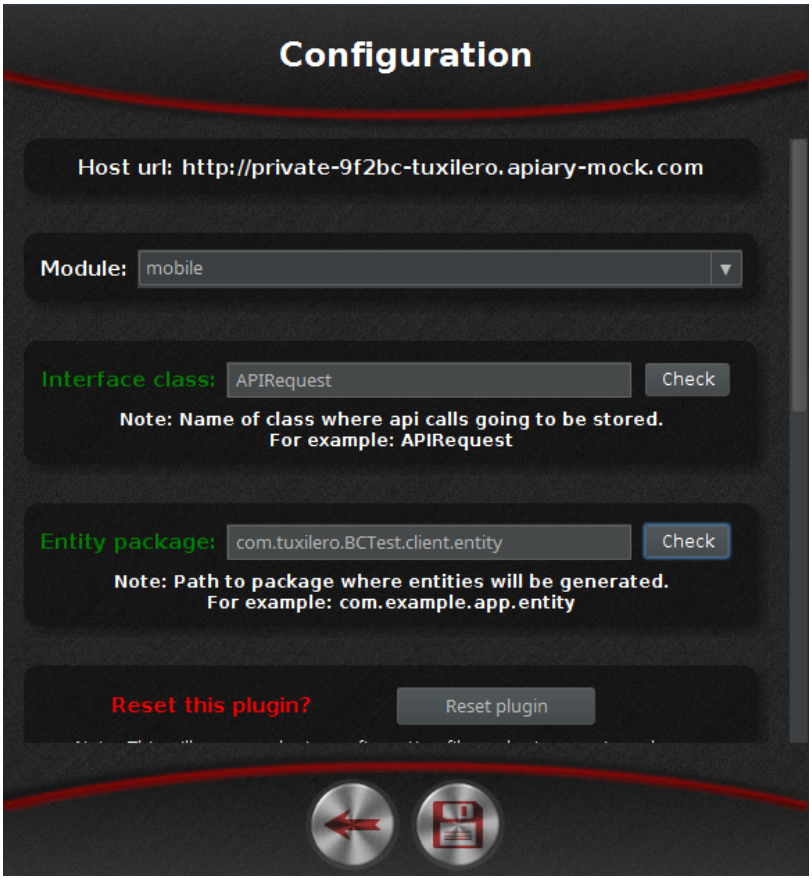


Figure 4.7: Part of Configuration screen (Figure 4.7) with correctly set Interface class and Entity package.

4.3.9 Configuration Screen

There are two UI elements in the `middlePanel` for purely informational purpose. First UI element is `JLabel` containing the host URL parsed from the API Blueprint, the second one is `JTextArea` at the bottom of the `middlePanel`, which contains example text of what could the `APIManager` class look like. `APIManager` is a class that a developer using the plugin should implement in his project, as communicator for Retrofit framework.

To ensure proper functioning of the plugin, correct module must be selected. `ProjectManager` is looking for specific class names and entity packages in a specific module. This module can be selected using `JComboBox` which contains names of all the modules in the project.

Interface class also has to be set correctly. All the request methods for Retrofit framework have to be stored in this Interface class, so that the plugin would know where to find them. The Entity package has to be set the same way in order for the plugin to know where it should look for entities, and where the plugin will store generated entities. By clicking the check buttons the developer can easily check whether the interface class or the entity package exists. The name of the request method or the entity will change color to green in case it is implemented correctly. In other cases it will change color to red, and show a dialog containing problems description once it's clicked. This verification works the same way as checking for problems in the Main screen (4.3.8).

Sometimes developer needs to reset plugin, and start from scratch. To make this as easy as possible there is a reset plugin button which deletes the configuration file, cleans preferences and displays the Welcome screen (Figure 4.3) again.

Clicking the save button sets values into the `ConfigurationEntity`, and saves them using `ConfigPreferences` class into the abm_config.xml file described at (4.3.2). Default values displayed in Configuration screen (Figure 4.7) are also loaded from this configuration file.

Figure 4.8: First Implementation screen (Figure 4.8) containing information about POST request on `/note` URI.

4.3.10 First Implementation Screen

The First Implementation screen (Figure 4.8) shows basic information about the selected request. In case the request is already implemented, this screen will be preloaded with data from the `TreeNodeEntity`. Things to fill are the ones that plugin cannot automatically parse from the API Blueprint. This stands for the Method name that will represent this request, and all names for the Request and Response entities. There is also one `JCheckBox` which represent a choice whether the request should be implemented as synchronous or asynchronous.

Information in `TreeNodeEntity` comes already parsed from the Main screen (Figure 4.6), so the request and response example text is only displaying data from this entity, and `JTextField` for the entity names is also generated based on `BodyObjectEntity` list in the `TreeNodeEntity`.

After clicking the next button plugin checks all provided names for invalid characters, and in case of any problem displays an alert dialog, otherwise shows the Second Implementation screen (Figure 4.9).

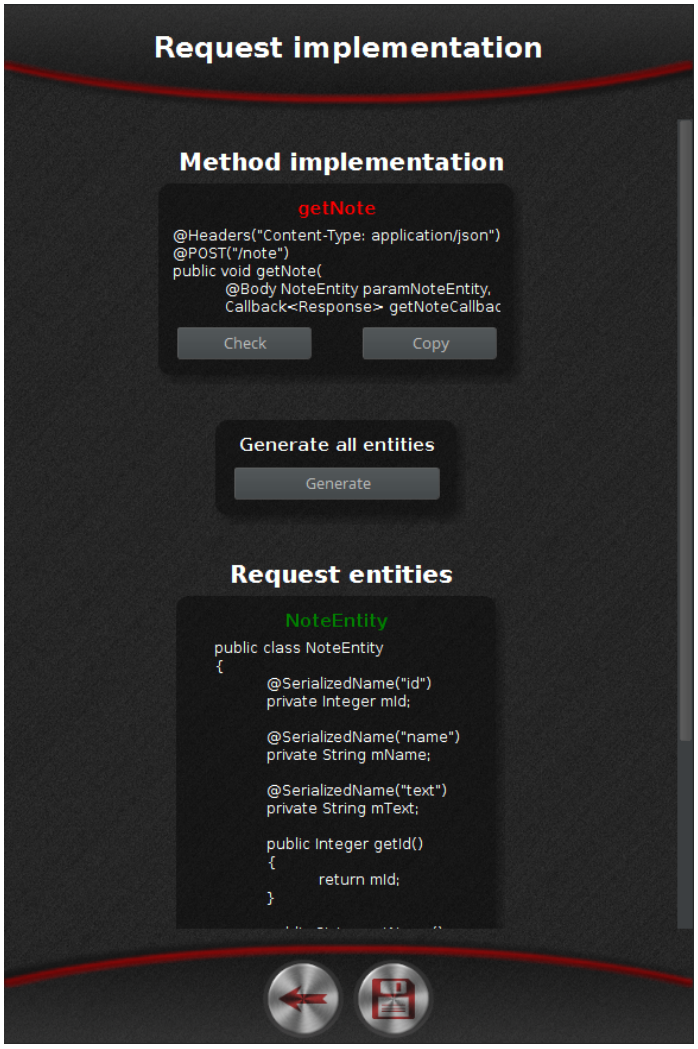


Figure 4.9: Second Implementation screen displaying wrongly implemented `getNote` method, and correctly implemented `NoteEntity` entity.

4.3.11 Second Implementation Screen

Second Implementation screen (Figure 4.9) receives `TreeNodeEntity` already enriched with all the necessary information. The purpose of this screen is to generate example code. Developer is given the possibility to generate entity files automatically, or copy the code manually, into the right files. Developer can also verify that the code he wrote or pasted into the file is correct.

Two methods take care of this. `generateMethodExample` and `generateEntityExample` generate example code. These methods create new empty string, and append more and more lines into it, based on the methods' logic.

The logic of `generateMethodExample` method, is to generate string following these steps:

1. Add all request headers (eg. "`@Headers('Content-Type: application/json')`")
2. Add request URI with correct method (eg. "`@POST('/note')`")
3. Add a method with correct return type and name (eg. "`public void getNote()`")
4. Add parameters into this method if any:
   (a) Add Path and Query.
   (b) Add Body, with correct Request entity.
   (c) If the method was configured as asynchronous on the First Implementation screen (Figure 4.8), Callback with correct Response entity type will be added as the last parameter.

The logic of `generateEntityExample` method is to generate a valid implementation of the whole class into a string. This string starts with class definition with the provided entity name. To fill this class, method loops through each `BodyVariableEntity`, to generate variable names with annotations (for example: "`@SerializedName('id')`" and "`private Integer mID;`"). After that the method loop all those `BodyVariableEntity` again, to generate getters (for example: "`public Integer getID(){ return mID; }`").

If the developer clicks on the generate file button, `generateEntityFile` method will generate new file in entity package specified in Configuration screen (Figure 4.7). If the file already exists, plugin won't overwrite this file, and notify the user that the file already exists.

Clicking the check button will call `checkMethodForProblems` or `checkEntityForProblems` from `ProjectManager` class, which will return the `ProblemEntity` list, displaying it using a dialog, as described in the chapter about the Main screen (4.3.8).

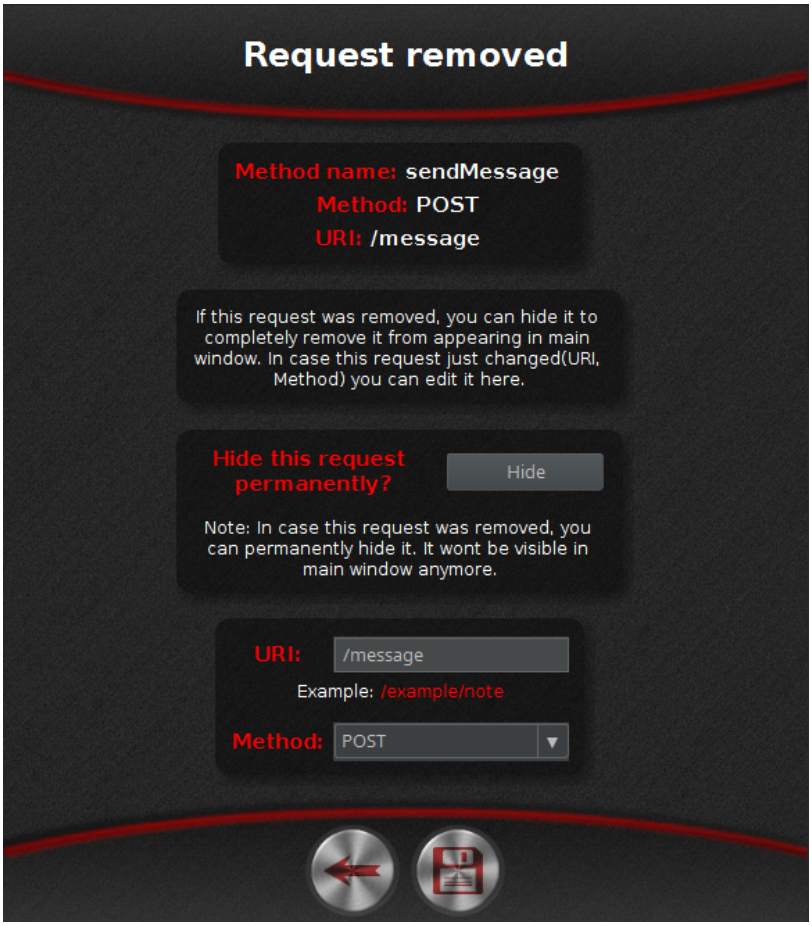


Figure 4.10: Removed screen displaying information about removed request, with `JComboBox` and `JTextField`.

4.3.12 Removed Screen

Plugin uses HTTP method (POST, GET etc.) and URI address, to clearly identify a request. If one of those values changes in the API Blueprint or the request is removed from the API Blueprint, and the request is already implemented in user code, it will be recognized as removed from the API Blueprint, and no longer needed to be implemented in the code.

In case the request has really been removed, developer can hide it permanently. This can be done clicking on hide button, which will mark `TreeNodeEntity` as removed, and will no longer display in `JTree` in the Main screen (Figure 4.6). But if the request appears again in API Blueprint, the `TreeNodeEntity` will be automatically marked as existing, and will be visible again.

In case the request hasn't been removed, and the URI or Method has been changed, developer can manually change the URI and Method on this screen. Which will save the `TreeNodeEntity` with new URI and Method.

Chapter 5

Testing and Evaluation

5.1 Testig Method

Three experienced Android developers working for STRV company for plugin testing were picked. All three developers were provided with a basic information about the plugin usage, and the usage video[3]. Also, these three developers received basic training on how API Blueprint works.

Testing was focused on time efficiency because the main aspect of this thesis is to save developer's time by providing him with detailed information about API Blueprint implementation and code examples. For this purpose these two aspects of plugin were tested:

- How much time developer saves by using the plugin
- How well can plugin announce what have changed

These three application developers had to implement a specific testing API Blueprint in the prepared Java code. First, they had to implement methods for each request, and implement then all the necessary entities.

After completing this task, changes were made in the API Blueprint. These changes have been the same for all three developers. Testers then had to find all changes, and modify their code, so that it would be correct again.

5.2 Testing Data

API Blueprint that was used for testing can be found in appendix 7. All three testers had to implement this API Blueprint.

After completing their implementation, the following changes was made in the API Blueprint:

- Removed "Note create" resource.
- Changed data type from Double to Integer in one of Weather entities.
- Added new Boolean element into the Message Model

[3] https://www.youtube.com/watch?v=ZHRyQUN5bUc

5.3 Evaluation

The results of API Blueprint implementing test was the following (time is measured in minutes):

User	Manual implementation time	Implementation time using plugin
Developer 1	17:35	3:06
Developer 2	17:42	4:14
Developer 3	16:33	3:21

From those results can be seen that the implementation time using plugin is almost five-times faster. This was really surprising since it was unexpected that the impact could be so big.

The results for the second part of the testing (recognize and implement all changes correctly) are the following:

User	Manual implementation time	Implementation time using plugin
Developer 1	3:45	2:14
Developer 2	4:12	1:53
Developer 3	5:54	2:27

In this case, the difference in the amount of time the implementation took each developer was really big. The implementation time quite depended on how well the developer remembered the original API Blueprint before the changes were made.

Either way, the difference in times between manual implementation and implementation using plugin wasn't that big. It is believed that the time differences would be bigger with an extensive API Blueprint.

All three testers liked the graphic design of the plugin, and especially the possibility of generating all the entities with just one click, which saved them the most time during the first part of the test. All three developers also stated that creating this plugin was a good idea. On the other hand, they agreed on the fact that in next versions the plugin should support more than just the retrofit framework.

Chapter 6

Conclusion

As has been asserted in chapter 5, this solution can save hours, and on extensive APIs event days of work, just by generating example code for requests and all necessary entities for those requests. Plugin will also save developers a lot of time by notifying them about what and where has changed in the API. The more robust the API is, the more of the developer's time it will save.

There are several versions of this plugin released. To this day (13.5.2014) the current version is 2.1.0, and the plugin in JetBrains repository (https://plugins.jetbrains.com/plugin/7713) counts total 240 downloads.

Based on user reviews of older versions, functionality for generating entities or resetting the plugin has been added in version 2.0.3. Also, several bugs were fixed in later versions. As all changes have been committed to GitHub, the source code and all commits can be found at https://github.com/Tuxilero/ABM.

Next versions of plugin may add support for another network framework other than Retrofit, as well as support for another data type besides JSON. There is also a lot of special API Blueprint syntax that could be implemented.

Bibliography

[1] Roy Thomas Fielding. Architectural styles and the design of network-based software architectures [online]. http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf, 2000.

[2] Roy Thomas Fielding. Rest apis must be hypertext-driven [online]. http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008-10-20.

[3] Apiary Inc. About apiary [online]. https://apiary.a.ssl.fastly.net/assets/files/apiary-press-kit-29b1ca7e9b466854.zip, 2014.

[4] Apiary Inc. Api blueprint tutorial [online]. https://github.com/apiaryio/api-blueprint/blob/master/Tutorial.md, 2015.

[5] Google Inc. Android studio overview [online]. http://developer.android.com/tools/studio/index.html.

[6] JetBrains s.r.o. Intellij idea features [online]. https://www.jetbrains.com/idea/features/.

Chapter 7

Testing API Blueprint

```
FORMAT: 1A
HOST: http://private-9f2bc-tuxilero.apiary-mock.com

# Mock API
This is a test blueprint for my thesis

# Group Weather

## /weather{?country,state}

### weather [GET]
Get weather from server

+ Parameters
    + country (string) ... Country tag, for example: "us", "cz", "uk".
    + state (string) ... State.

+ Response 200

    + Headers

            Content-Type: application/json

    + Body

            {
                "country" : "cz",
                "city" :
                [{
                    "base" :
                    {
                        "temp" : 24,
                        "temp_min" : 14,
                        "temp_max" : 26,
                        "humidity" : 65,
                        "pressure" : 989
                    },
                    "wind" :
                    {
                        "speed" : 1.16,
                        "deg" : 312
                    },
                    "clouds" :
                    {
                        "value" : 2,
                        "text" : "Sunny day"
                    }
                }]
            }
```

```
50
51  # Group Notes
52
53  ## Note [/note]
54  Create new note.
55
56  + Model (application/json)
57
58      JSON representation of Note.
59
60      + Body
61
62              {
63                  "id" : 666,
64                  "name" : "Test note",
65                  "text" : "Some text inside note"
66              }
67
68
69  ### createNote [POST]
70  Create new Note
71
72  + Request
73
74      [Note][]
75
76  + Response 201
77
78  ## Note [/note/{id}]
79  Get old note by ID.
80
81
82  ### getNote [GET]
83  Get note by ID
84
85  + Parameters
86      + id (integer) ... ID of note to retrieve.
87
88  + Response 200
89
90      [Note][]
91
92  # Group Messages
93
94  ## Message [/message]
95  Send message.
96
97  + Model (application/json)
98
99      JSON representation of message.
100
101     + Body
102
103             {
104                 "from" : "fromName",
105                 "to" : "toName",
106                 "message" : "messageText",
107                 "timeStamp" : "date"
108             }
109
110
111 ### sendMessage [POST]
112 Send Message
113
114 + Request
115
116     [Message][]
117
```

```
+ Response 201


## Message [/messages/{user}]
Receive messages for user.


### getAllMessages [GET]
Get all messages of user

+ Parameters
    + user (string) ... Name of user. For example: "mySuperUsername".

+ Response 200

    + Headers

            Content-Type: application/json
    + Body

            {
                "MessageList" : [{
                    "from" : "fromName",
                    "to" : "toName",
                    "message" : "messageText",
                    "timeStamp" : "date"
                }]
            }
```