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

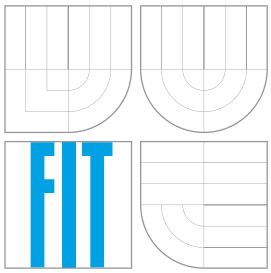
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ HERMANN

BRNO 2015

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

GENEROVÁNÍ A ÚDRŽBA TŘÍD JAVA NA ZÁKLADĚ APIARY BLUEPRINTU

GENERATION AND MAINTENANCE OF JAVA CLASSES BASED ON APIARY BLUEPRINT

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

LUKÁŠ HERMANN

VEDOUCÍ PRÁCE
SUPERVISOR

doc. Ing. HEROUT ADAM, Ph.D.

BRNO 2015

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.

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.

Klíčová slova

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

Keywords

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

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 11, 2015

Poděkování

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

Contents

1 Introduction **3**
- 1.1 Motivation . . . 3
- 1.2 Structure of the Document . . . 4

2 API, REST API, Apiary and API Blueprint **5**
- 2.1 REST API . . . 5
- 2.2 Apiary . . . 6
- 2.3 API Blueprint . . . 7

3 IDE and Plugin Development for InteliJ IDEA **9**
- 3.1 IntelliJ IDEA . . . 9
- 3.2 Android Studio . . . 10
- 3.3 Plugin and Plugin development for InteliJ IDEA . . . 10

4 Problem and solution using plugin **12**
- 4.1 Problem . . . 12
- 4.2 Suggestion of solution . . . 12
- 4.3 Implementation . . . 16
  - 4.3.1 Project structure . . . 16
  - 4.3.2 Plugin configuration . . . 17
  - 4.3.3 ProjectManager class . . . 17
  - 4.3.4 Plugin initialization . . . 18
  - 4.3.5 Welcome screen . . . 19
  - 4.3.6 Connect screen . . . 20
  - 4.3.7 Gradle screen . . . 21
  - 4.3.8 Main screen . . . 22
  - 4.3.9 Configuration screen . . . 24
  - 4.3.10 First Implementation screen . . . 26
  - 4.3.11 Second Implementation screen . . . 27
  - 4.3.12 Removed screen . . . 29

5 Testing and Evaluation **30**
- 5.1 Testig method . . . 30
- 5.2 Testing data . . . 30
- 5.3 Evaluation . . . 31

6 Conclusion **32**

A Testing API Blueprint **33**

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 the REST API. 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 request in the application code. Apiary is one of such services those provides a possibility of generating documentation based on API Blueprint, which is language based on 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 server to process, or to obtain data from it.

Let's assume that there are more developers working on a project. One of them is 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 any time. If he won't let's 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 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 from 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 discussed 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 plugin for IDE. This chapter includes problem description, solution design and description of implementation of each part of the plugin.

Testing 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 some unit (for example library, or service) which 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 in his dissertation thesis [1] stated that REST (Representational State Transfer) is 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 example API Blueprint was specified `FORMAT` with 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, there one needs to provide a resource name and URI used to access this resource, placed inside square bracket at the end of the heading. Resource can have a description placed on the new line right after 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 unrelated 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 to 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.

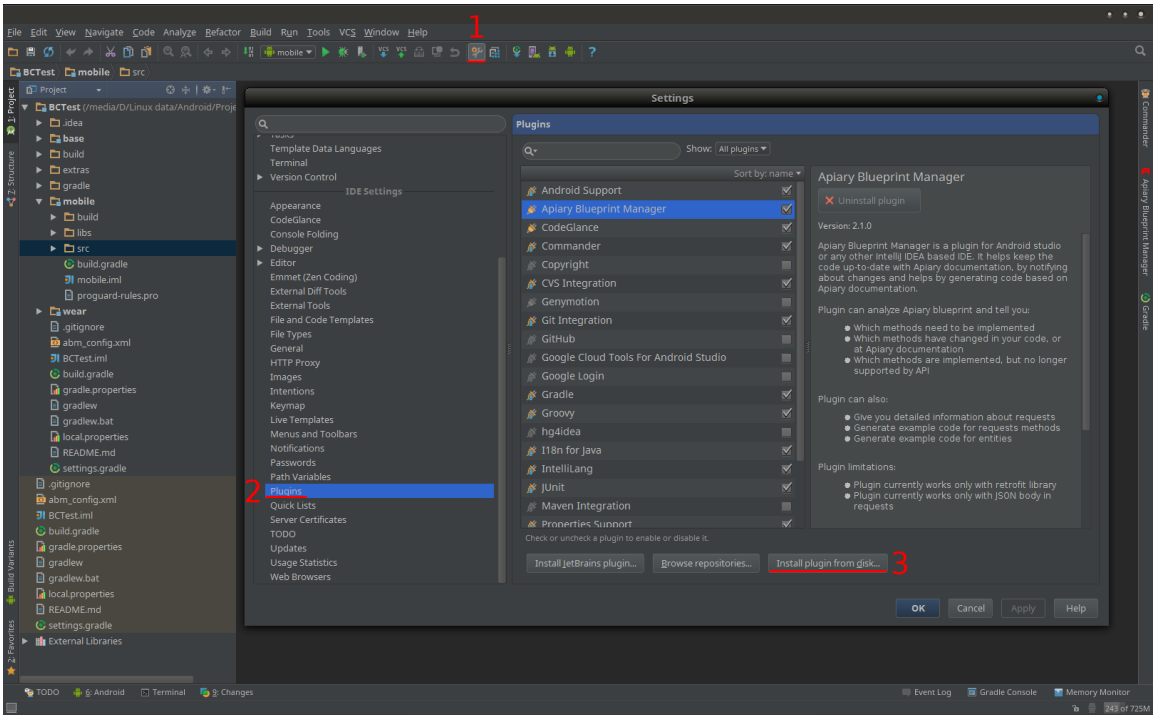


Figure 3.1: Android studio with three easy Plugin installation steps.

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 Plugin 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 be also 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 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 should automatically create plugin.xml file in the META-INF folder, located in the root directory of currently used project module. This file contains information about the plugin, like name, version, vendor or 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 <plugin> 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 application developer with implementation of the basic action 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 realize 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 some entity structure. Plugin has to take this representation of data, and compare it to the code in the project. After comparing data, plugin has to show differences of what has changed, what is not implemented or 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. As for user interface design, it is wise to implement this type of plugin as a `ToolWindow` class.

API Blueprint can be obtained from the official Apiary site, by providing 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 that is very easy to use, and can make HTTP request as GET, 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 can be found 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 sending to the parser, and receiving the response.

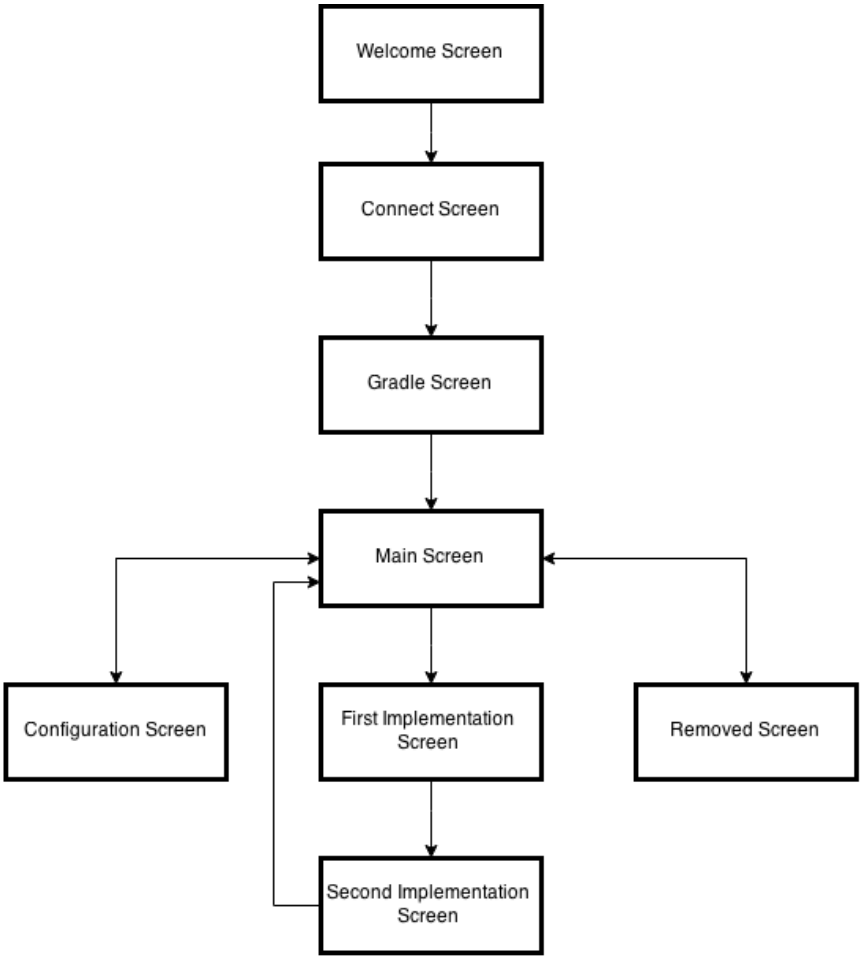


Figure 4.1: Flow diagram of all plugin screens.

There will have to be a method for analyzing the API Blueprint, this method has to analyze 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. Those PSI classes provide easy verification of 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..

From this analysis the plugin will generate a list of items indicating the state of 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, plugin can generate example code for the request method, and all 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 Retrofit framework.

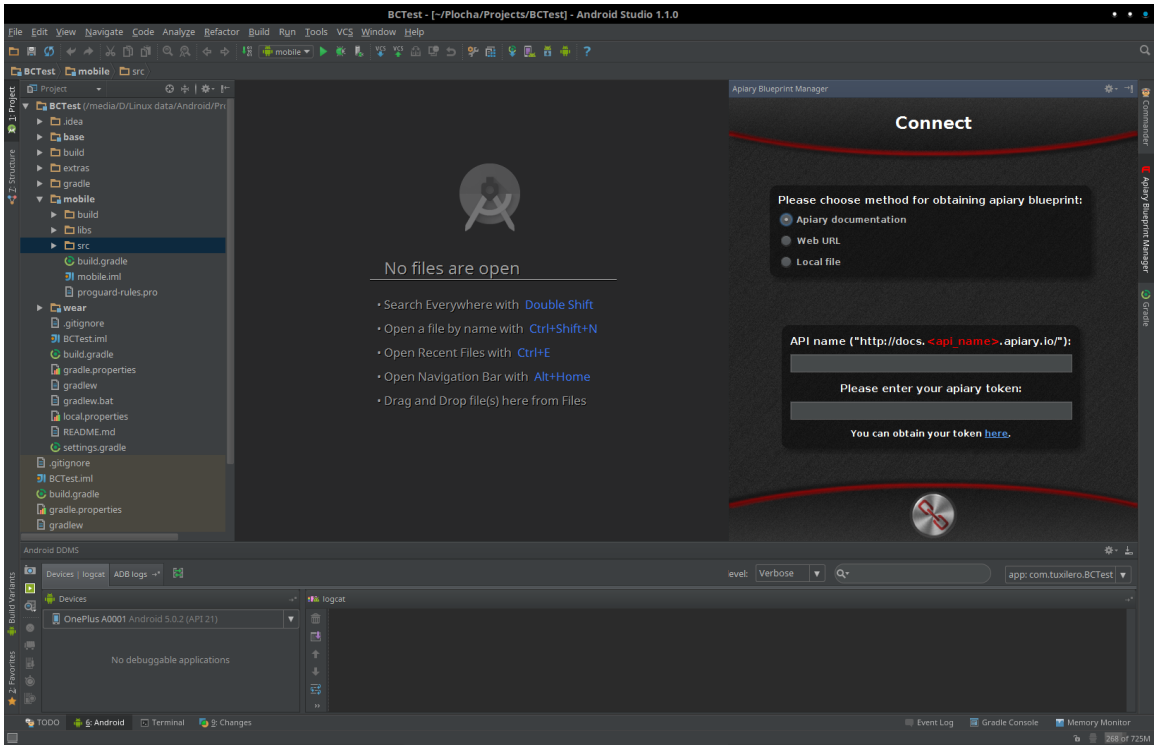


Figure 4.2: ABM Plugin with Connect screen (Figure 4.4) opened in Android Studio, showing the color tuning of Plugin with Android Studio.

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 dark theme in their IDE, I've decided to choose dark grey color, combined with red elements and white texts which fit perfectly for this dark theme as can be seen on (Figure 4.2). Black boxes that can be seen on image 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 Main screen (Figure 4.6), which indicates that API Blueprint is being downloaded and refreshed. GIF images were used for those 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 plugin content size to the resolution of a screen. Using fixed sizes of font and spaces resulted in huge buttons, text, etc. in 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 contain a 1px border. The colors in the border determine if a piece is static, it stretches, or it 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. The biggest problem is with animated buttons using GIFs as source. 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 resized the normal way, 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

Project in Java can have for example this structure:

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

Plugin project sources are structured to a few packages:

- Entity – contains all entity classes for the plugin project. It also contains another two packages 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 work 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 some constants, like plugin version, temp file names, and switches for debugging and logging.

Plugin project resources are structured to these packages:

- Drawables – all images used in plugin project.
- META-INF – plugin.xml file with plugin configuration and information about this 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 there is a problem detected, it is added to this list, which is then returned as an output of those two methods.

Method `checkMethodForProblems` takes `TreeNodeEntity` as 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 create and return new `ProblemEntity` informing about his.
2. Compare method headers with headers specified in API Blueprint.
3. Compare method Method (GET, POST, eg.) to API Blueprint method.
4. Check method return type.
5. Get `PsiParameterList` of implemented method, `ParametersEntity` list from `TreeNodeEntity` and `BodyObjectEntity` list from `TreeNodeEntity`:
   (a) Loop through all `PsiParameterList` items of implemented method.
   (b) If current item matches any 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 those list 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 exist in `BodyObjectEntity` list of `TreeNodeEntity`.
4. If it exists, check the variable return 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 `toolWindow` element. This means that plugin will appear in tool section of IDE, and will behave as a tool windows as can be seen on (Figure 4.2). This element has the attribute `factoryClass` which accepts class as value. This class is initialized right after clicking the plugin button in IDE. In this plugin it is 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

At the beginning of every screen, there is a `initLayout` method in which is whole screen rendered. It is pretty same for every screen, the logic stays the same, only difference is the content of `middlePanel`. First, is created `JBackgroundPanel` class with background image and appropriate type `BACKGROUND_REPEAT`, and it is set as `toolWindow` content. Instance of `toolWindow` have to be transmitted from screen to screen. There is also need to set MigLayout to instance of `JBackgroundPanel`.

Layout is composite from three rows and one column. Top and bottom also contains `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 screen displayed.

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

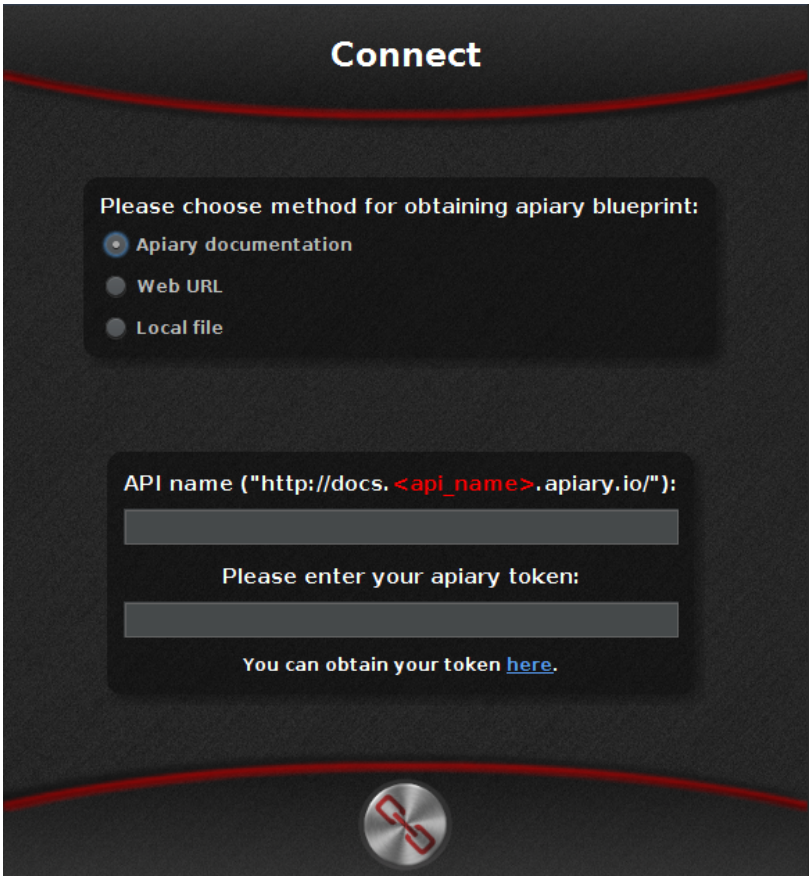


Figure 4.4: Connect screen with three JRadioButtons and JTextFields.

4.3.6 Connect screen

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

There are three radio buttons, which switch between connection method. After switching to another one, the bottom box changes based on which radio button was selected. This is implemented by using JPanel with `CardLayout`. There are three normally layouted `JBackgroundPanels`, that are added to this `CardLayout`, and then they can be easily switched using show method of the `CardLayout`.

After clicking the connect button, plugin start processing data based on which connection method is selected. The logic is pretty same in all three cases:

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

2. After obtaining API Blueprint there is need to check if the API Blueprint is valid, and there are no errors in it. For this plugin call Network class method `isBlueprintValid`.

3. In case everything is OK, and API Blueprint is valid, plugin save API Blueprint into

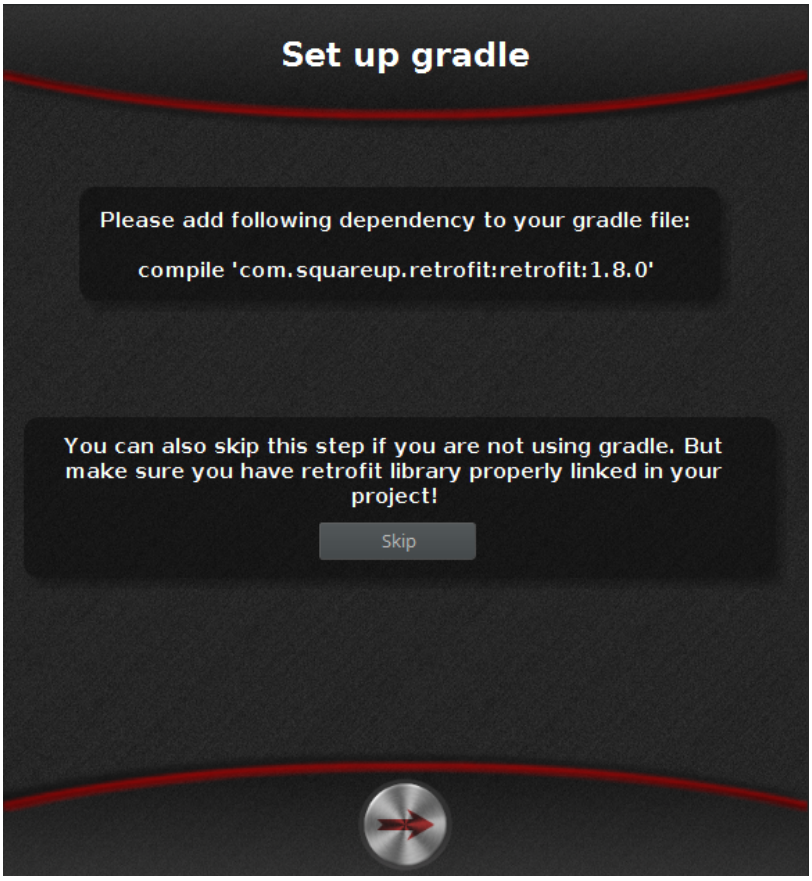


Figure 4.5: Gradle screen with information and skip button displayed.

temp file using Utils class method `saveStringToTmpFile`, and set Preferences values for temp file location, connection type, path, and in case of Apiary documentation connection type it also save apiary token.

4. After saving and procesing all information, plugin proceed to Gradle screen (Figure 4.5).

4.3.7 Gradle screen

This screen is really simple. All it does is printing information that developer have to use Retrofit framework in his project, to make it work correctly. There is also button for skipping this step, so the plugin won't check if Retrofit framework is added to Gradle because not everyone uses Gradle in his project.

When next button at the bottom of the screen is clicked, plugin check `gradle.build` file for Retrofit framework occurrence, using Utility class method `isGradleWithRetrofit`. In case it is missing, he displays dialog informing user about it, otherwise plugin display 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. There is implemented the main logic of the whole plugin. As for layout, there is a change in top panel, where are located two buttons and image. Left button displaying dialog with information about plugin, and right button displaying plugin configuration. The middle image displaying state of code. There are three states:

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

`middlePanel` on this screen contains JTree class for displaying errors and differences between code and API Blueprint. For making its text colored and different sized, there is implemented `ABMTreeCellRenderer` class in which text is resized and colored based

on `TreeNodeEntity` type. Data for this JTree are generated using `initTreeStructure` method which construct tree items based on `TreeNodeEntity` list.

`TreeNodeEntity` list is list of entities that store 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 if the node is visible or not.

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

1. Load API Blueprint from temp file in case of just displaying, or download it again in case of reloading using reload button. This mean that if user just change screens, and return on Main screen (Figure 4.6), it won't download API Blueprint again, and use old stored one, and to check and download actual API Blueprint, he has to click the refresh button.

2. Right after loading API Blueprint there is a need to convert it to JSON. Which is done by using Network class method `requestJSONFromBlueprint`. This method takes the API Blueprint text, and send it to official API Blueprint parser http://api.apiblueprint.org/parser. This parser than return JSON representation of API Blueprint.

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

4. Having this object plugin call `analyzeBlueprint` method, which takes the `ABMEntity`, and analyze it, converting it to list of `TreeNodeEntity`.

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

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

`analyzeBlueprint` method converts `ABMEntity` to `TreeNodeEntity` by taking only information plugin need, and everything else throwing away. Also, it parse Request and Reponse example JSON data, to `BodyObjectEntity` because plugin need this data later to be able to generate example entities to save these Request and Response data. This parsing is done by using GSON library, by parsing data to Object class that is the root of the class hierarchy. Parsing JSON data this way, let plugin 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 actual project code state. First, this method check using `ProjectManager` class if entity package and interface class are configured properly. If not, the output of this method will be `TreeNodeEntity` list containing only one, or two items containing error text that entity package and/or interface class is not configured properly. After that method set correct `TreeNodeType` to the `TreeNodeEntity`, so it is displayed correctly in JTree. This is done by using `ProjectManager` class, which can tell if there is a difference between local project code and API Blueprint.

Selecting item in the JTree run action depending on type of `TreeNodeEntity`:

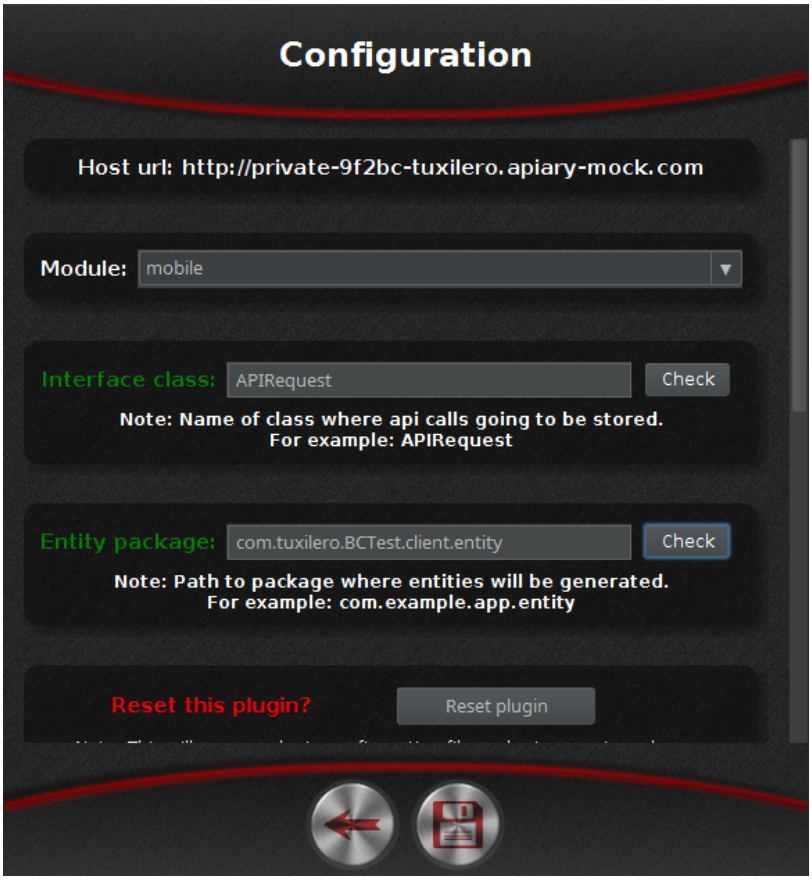


Figure 4.7: Part of Configuration screen (Figure 4.7) with correctly set Interface class and Entity package.

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

4.3.9 Configuration screen

First item in `middlePanel` is JLabel containing host URL parsed from API Blueprint for information purpose. Also, for information purpose there is JTextArea as a last item in `middlePanel`, which contains example text of how could `APIManager` class looks like. `APIManager` is a class that should developer using the plugin implement in his project, as communicator for Retrofit framework.

To ensure proper functioning of the plugin, correct module must be selected because `ProjectManager` is looking for specific class names and entity packages in concrete module. For this plugin use JComboBox which contains all modules names in the project.

There is also need to set Interface class, in which are requests methods stored, so plugin will know where to find them. Same way need to be set entity package, where will be plugin

looking for entities, and where will plugin store generated entities. For this there are check buttons, which check if interface class, respectively entity package exist, and color the text to red or green depending if it is set correctly.

Sometimes user need to reset plugin, and start from beginning. For this there is a reset plugin button which delete configuration file, clean preferences and display Welcome screen (Figure 4.3) again.

Clicking the save button set values into `ConfigurationEntity`, and save it using `ConfigPreferences` class into `abm_config.xml` file. 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

On the First implementation screen (Figure 4.8) is displayed basic info for user, about selected request. In case the request is already implemented, this screen will be preloaded with data from `TreeNodeEntity`. Things to fill are those that plugin cannot automatically parse from API Blueprint. This stands for Method name that will represent this request, and all names for Request and Response entities. There is also one JCheckBox which represent if the request should be implemented synchronous or asynchronous.

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

Clicking the next button, plugin verify all filled names for invalid characters, and in case of some problem displays alert dialog, otherwise shows Second Implementation screen (Figure 4.9).

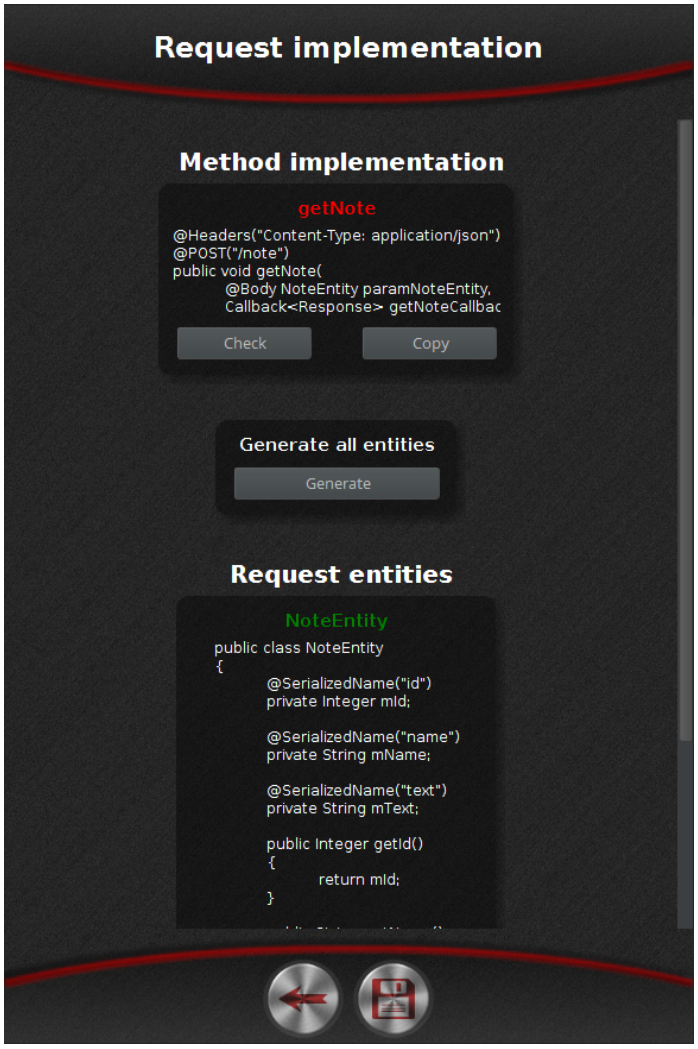


Figure 4.9: Second Implementation screen displaying wrongly implemented getNote method, and correctly implemented NoteEntity entity.

4.3.11 Second Implementation screen

Second implementation screen (Figure 4.9) receive `TreeNodeEntity` already filled with all necessary information, so whole point of this screen is to generate example code, provide user with possibility to generate entities files automatically, or copying text manually, in to correct files, and let user check if the code he wrote or paste into the file is correct.

To do this, there are two methods `generateMethodExample` and `generateEntityExample` for generating example code. These methods create new empty string, and appending more and more lines into it based on the methods logic.

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

1. Add all request headers (for example: "`@Headers('Content-Type: application/json')`")
2. Add request URI with correct method (for example: "`@POST('/note')`")
3. Add method with correct return type and name (for example: "`public void getNote()`")

4. Add parameters into this method if any:

   (a) Add Path and Query.

   (b) Add Body, with correct Request entity.

   (c) Add Callback, if method was set as assynchronous on First Implementation screen (Figure 4.8). The Callback must have set Response entity correctly.

The logic of `generateEntityExample` method, is to generate string of whole class. This string starts with class definition with appropriate entity name. To fill this class method do for each loop for 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 user click on generate file button, `generateEntityFile` method will generate new file in entity package specified in Configuration screen (Figure 4.7). If the file already exist, plugin won't overwrite this file, and notify user about that the file already exist.

Clicking the check button will call `checkMethodForProblems` or `checkEntityForProblems` from `ProjectManager` class, which will return the ProblemEntity list, displaying it using dialog.

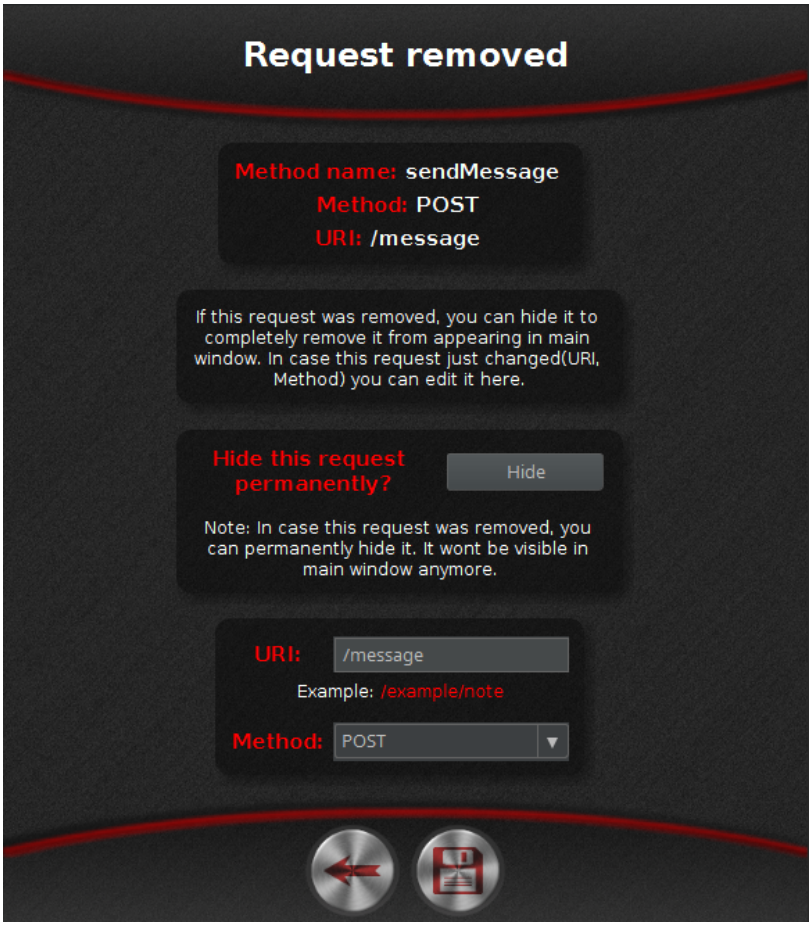


Figure 4.10: Removed screen displaying information about removed request, with JComboBox and JTextField.

4.3.12 Removed screen

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

In case the request is really removed, user 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 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 wasn't 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

Two aspects of plugin was tested:

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

For plugin efficiency testing purpose was picked three application developers, which had to implement specific testing API Blueprint in prepared Java code. First, they had to implement methods for each request, and implement all necessary entities.

After completing this task, there was made some changes in API Blueprint. The changes have been same for all three users. Tester then had to find all these changes, and modify his code, so it would be correct again.

5.2 Testing data

API Blueprint that was used for testing can be found in appendix A. This is the API Blueprint that had all three tester implement.

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

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

5.3 Evaluation

The results of implementing test API Blueprint was 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 large.

The results for the second part of the testing (recognize and implement all changes correctly) are 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 implementation time of the developers 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 wasn't so big. It is believed that the time differences would be bigger on extensive API Blueprint.

Chapter 6

Conclusion

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

There were several versions of this plugin released. To this day (7.5.2014) the current version is 2.1.0, and this plugin in JetBrains repository (https://plugins.jetbrains.com/plugin/7713) counts total downloads of 219.

Based on users review of older versions, there were added generate entity functionality, and functionality for resetting plugin, in version 2.0.3. Also, several bugfixes was implemented in later versions.

As all changes have been commited on GitHub, the source codes and all commits can be found at https://github.com/Tuxilero/ABM.

Next versions of plugin could add support for another Network framework like Retrofit, also adding support for another data type, not just JSON would be great. There is also a lot of special API Blueprint syntax that could be implemented too.

Appendix A

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"
                }]
            }
```

Bibliography

[1] Roy Thomas Field-
ing. 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/.