

# Fhlintstone Developers Guide

## Table of Contents

1. Introduction .....	2
2. Getting Started .....	3
3. Toolkit .....	4
3.1. HAPI FHIR .....	4
3.2. Logging .....	4
3.3. Dependency Injection .....	4
3.4. Lombok .....	4
3.5. JavaPoet .....	5
3.6. picocli .....	5
3.7. Test Automation .....	5
3.8. Spotless .....	5
3.9. AsciiDoctor .....	5
4. How-To.....	6
4.1. Installing SUSHI .....	6
4.2. Installing Firely Terminal .....	7
4.3. Creating Test Packages .....	7
5. Glossary .....	9

---

# 1. Introduction

Fhlintstone is a Java code generator to support [HL7 FHIR](#) developers using the popular [HAPI FHIR](#) framework. Based on one or multiple [NPM Packages](#) containing the various definitions, it can be used to

- generate Java enums for [ValueSets](#)
- generate Java classes to implement [custom resource and extension classes](#) for [StructureDefinitions](#) and
- facilitate [handling profiles and extensions](#) by providing builders tailored to the specific FHIR adaptations described by the package contents.

This document is intended to help developers who want to contribute to the Fhlintstone software itself. If you only want to use Fhlintstone and not extend or debug the tool yourself, this is not the document you're looking for - you should find everything you need in the Users Manual.

This document assumes you're already familiar with the basics of Java development - if you're not, Fhlintstone is probably not a good project to get started due to its high complexity. The same holds true for [HL7 FHIR](#) - you should have at least some working FHIR knowledge and be able to read and understand [ValueSet](#) and [StructureDefinition](#) resources.

## 2. Getting Started

In order to work with the Fhlintstome sources, you only need a few basic requirements:

- a Java JDK in Version 21 or newer,
- a working Maven installation,
- git and
- an IDE of your choice.

With this, you should be able to check out the source code from the [main repository](#) or your own fork of this repository. The project can be built with a simple `mvn clean install` or from your preferred IDE. The first build process will - as usual - probably take some time because a number of dependencies will be installed. Once this has happened, a complete build process will still take a few minutes due to the large number of complex automated tests.

Once you have completed these basic steps, you should be able to contribute to the Java implementation. It is recommended that you familiarize yourself with the [toolkit](#) used by Fhlintstone and take a look at the architectural documentation, which is distributed as a separate document. It is also strongly recommended that you [install SUSHI](#) so that you are able to contribute to the test data as well as the Java source code.

If you want to run a SonarQube check, we recommend disabling these rules:

ID	Name	Reason
<b>java:S1135</b>	Track uses of “TODO” tags	We use a TODO with the issuer ID to mark the places in the code where something needs to be done.
<b>java:S4738</b>	Java features should be preferred to Guava	We prefer to use an ImmutableList rather than a List. This makes it clear that the values cannot be changed.
<b>java:S110</b>	Inheritance tree of classes should not be too deep	We are extending the HAPI FHIR libraries, so we have to inherit from many classes.
<b>xml:S1135</b>	Track uses of "TODO" tags	We use a TODO with the issuer ID to mark the places in the code where something needs to be done.

## 3. Toolkit

This section gives a brief overview of the external components and libraries that you should be familiar with when working on Fhlintstone.

### 3.1. HAPI FHIR

Since Fhlintstone generates code that is based on [HAPI FHIR](#), it is obvious that you should be familiar with at least the following sections of the HAPI documentation:

- [Getting Started](#)
- [Working with the FHIR model](#)

### 3.2. Logging

Fhlintstone uses [SLF4J](#). The logger should be provisioned using [Lombok](#) by using the annotation [@Slf4j](#). Please make sure you add `logger.entry(...)` and `logger.exit()` for all but the most trivial methods.

### 3.3. Dependency Injection

Fhlintstone uses [Google Guice](#) in combination with [Eclipse Sisu](#) for dependency injection. Note that due to the Sisu version being currently used, you will still need to use the old annotations like `javax.inject.Named` - the newer annotations like `jakarta.inject.Name` are not yet supported (see also [sisu issue #92](#)),

### 3.4. Lombok

[Project Lombok](#) is used to reduce the amount of boilerplate code required. You should familiarize yourself with the following annotations that are currently being used:

- stable features
  - `@Getter` and `@Setter` with `@AccessLevel`
  - `@EqualsAndHashCode`
  - `@NoArgsConstructor`
  - `@Data`
  - `@Builder` and `@Singular`
  - `@Slf4j`
- experimental features
  - `@SuperBuilder`

Please be cautious when using new experimental features - always check the support status and possible stabilization issues in the Lombok documentation.

## 3.5. JavaPoet

Flintstone uses [JavaPoet](#) to generate the source code. Since this is a very crucial part of the entire operation, you should read the entire JavaPoet documentation at least once (it's not that long).

## 3.6. picocli

[picocli](#) is used to implement the command line interface.

## 3.7. Test Automation

Flintstone uses [JUnit 5](#) with a number of extensions and additions:

- [Mockito](#) to generate mock objects
- [AssertJ](#) for a more fluent test code style
- [EqualsVerifier](#) to automate tests of `equals()` and `hashCode()`
- [ArchUnit](#) to prevent violations of certain architectural principles

A few rules of thumb:

- Use either AssertJ (preferred for new tests) or classic JUnit assertions, but do not mix if possible (for better readability and maintainability).
- Use EqualsVerifier for all data-carrying classes (not required for process classes).
- Use unit tests (`...Test`) to test an isolated class (i.e. mocking every surrounding object) and use integration tests (`...IT`) to test the class with only partial mocking. Integration tests are commonly used to load the test packages and ensure that the component works correctly with the contents. This has turned out to be much more productive than attempting to mock large parts of the HAPI framework.

## 3.8. Spotless

Flintstone uses [Spotless](#) in combination with the lambda-friendly [Palantir Java Format](#) to keep a consistent code formatting. Please ensure to apply the style before contributing code to keep the noise level in the changes down. The code style is applied automatically in every build process, so if you perform a complete build and test run before creating a PR (which you most definitely should!), this will not pose a problem.

## 3.9. AsciiDoctor

This document and its siblings are built using [AsciiDoctor](#).

When editing the AsciiDoc sources, please keep each sentence to a single line to make PR reviews easier.

## 4. How-To...

This section contains descriptions of individual tasks that may become relevant when contributing to Fhlintstone.

### 4.1. Installing SUSHI

Fhlintstone uses [FSH](#) and [SUSHI](#) to generate [resources](#) for testing purposes.

Since SUSHI is not easily integrated into the main build process, the generated resources are kept in the main repository. Developers should therefore be able to generate the package contents locally, especially when [creating new test packages](#). This section describes the process to setup the infrastructure required to do so.

For macOS and Linux systems, it should be sufficient to follow the [official installation instructions](#). For Windows, the process can be a bit more involved. The following instructions assume that you are able to use the Windows Subsystem for Linux (WSL). This has the unfortunate downside of slowing SUSHI down, but the alternatives appear to be much more tedious to install.

1. Ensure that the base image is present (requires local admin privileges):

```
wsl --install -d Ubuntu
```

2. Install the WSL utilities:

```
sudo apt install -y wslu
```

3. Install the Java Runtime Environment:

```
sudo apt update
sudo apt install wget apt-transport-https gpg
wget -qO - https://packages.adoptium.net/artifactory/api/gpg/key/public | gpg
--dearmor | sudo tee /etc/apt/trusted.gpg.d/adoptium.gpg > /dev/null
echo "deb https://packages.adoptium.net/artifactory/deb $(awk -F=
'^VERSION_CODENAME/{print$2}' /etc/os-release) main" | tee
/etc/apt/sources.list.d/adoptium.list
sudo apt update
sudo apt install temurin-21-jdk
```

4. Install Ruby und Jekyll:

```
sudo apt install ruby-full build-essential zlib1g-dev
echo '# Install Ruby Gems to ~/gems' >> ~/.bashrc
echo 'export GEM_HOME="$HOME/gems"' >> ~/.bashrc
echo 'export PATH="$HOME/gems/bin:$PATH"' >> ~/.bashrc
```

```
source ~/.bashrc
gem install jekyll bundler
```

#### 5. Install SUSHI:

```
sudo apt install nodejs
npm install -g fsh-sushi
```

It is recommended to use [Visual Studio Code](#) with the [HL7 FHIR Shorthand Extension](#) to edit the source files.

## 4.2. Installing Firely Terminal

Flintstone uses [Firely Terminal](#) to convert [resources](#) from JSON to XML for testing purposes.

While [SUSHI](#) can be used to create resources ([instances](#)) that can be used to test the parsing and rendering process, it is only able to generate JSON files. Unfortunately, the HAPI implementations for XML and JSON have shown slightly different behavior in the past, so that we need to perform tests with both formats. To convert the instance resources contained in JSON files that were generated with SUSHI into XML files, Firely Terminal is used. Please follow the [installation instructions](#) on the package site. Note that these [alternative installation instructions](#) offer platform-dependent instructions, but currently still refer to an outdated version of the .NET SDK dependency. Make sure that you add the `fhir` executable to the search path.

## 4.3. Creating Test Packages

Flintstone uses [NPM packages](#) with custom-tailored resources to simulate specific conditions for testing purposes. New test packages for specific scenarios can be added whenever required; the process is described below. The process is relatively straightforward, although it is a lot easier to copy an existing package and modify it than to start from scratch. Note that this description does not cover any edge cases (like packages that are deliberately broken).

The test packages are stored in the submodule `flintstone-test-data` and packaged using the [Maven Assembly Plugin](#). The package files are then distributed to the various other submodules that use the packages using the [Maven Remote Resources Plugin](#). All of the actions described below assume that you're working in the base path `src/main/fhir` of this submodule.

**CAUTION:** Make sure that all URLs and names new resources contain the new package name and number. When copying from an existing package, ensure that you change the existing identifiers. URL collisions between packages can lead to very confusing errors in the integration tests.

**CAUTION:** Use an editor that knows how Makefiles work and that doesn't try to convert tabs into spaces and vice versa where it shouldn't.

1. Select the next package number available and document the package in `package-list.md` by adding a line to the table with the next number and a short description.
2. Create the package directory.

3. Create the input folder `input/fsh` and place whatever FSH source files required into this folder.
4. Create a new file `package.json` with the package manifest or copy and adapt an existing manifest.
5. Create a new file `sushi-config.yaml` or copy and adapt an existing SUSHI configuration.
6. Add a `Makefile` - preferably by copying an existing one. Make sure that all source files are listed in the `SOURCE_FILES` variable.
7. Extend the `PACKAGES` variable in the main `Makefile` to include the package in the semi-automatic build process.
8. Execute `make` in the base path `src/main/fhir` to automatically build the package contents.
9. Create a new `assembly descriptor` in the folder `assembly` (which resides on the same level as the package folders). By convention, the assembly descriptor is named after the package it belongs to. It is advisable to copy and adapt an existing configuration.

Note that you will have to execute the Maven build to ensure that the new package is distributed to the other submodules and can be used for local test executions.



## 5. Glossary

Term	Definition	Links
<i>CodeSystem</i>	<i>The CodeSystem resource is used to declare the existence of and describe a code system or code system supplement and its key properties, and optionally define a part or all of its content. Code systems define which codes (symbols and/or expressions) exist, and how they are understood.</i>	<a href="#">Resource CodeSystem - Content</a>
<i>Differential statement</i>	<i>Differential statements are one of two ways to describe the inner structure of a <a href="#">StructureDefinition</a>. Differential statements describe only the differences that they make relative to the base structure definition. In order to properly understand a differential structure, it must be applied to the structure definition on which it is based.</i>	<a href="#">Base Resource Definitions</a>
<i>Extension</i>	<i>Every element in a <a href="#">Resource</a> can have extension child elements to represent additional information that is not part of the basic definition of the resource. The use of extensions is what allows the FHIR specification to retain a core simplicity for everyone. To make the use of extensions safe and manageable, there is strict governance applied to the definition and use of extensions. Although any implementer can define and use extensions, there is a set of requirements that must be met as part of their use and definition.</i>	<a href="#">Extensibility</a>
<i>FHIR NPM Package</i>	<i>A set of <a href="#">Resources</a> bundled as a machine-readable archive file. Not to be confused with <a href="#">FHIR Package</a>.</i>	<a href="#">FHIR NPM Packages</a>
<i>FHIR Package</i>	<i>In the context of <a href="#">Profiling</a>, a group of related adaptations that are published as a group within an Implementation Guide. Not to be confused with <a href="#">FHIR NPM Package</a>.</i>	<a href="#">Profiling FHIR</a>
<i>FHIR Profile</i>	<i>A set of constraints on a <a href="#">Resource</a> represented as a <a href="#">StructureDefinition</a>.</i>	<a href="#">Profiling FHIR</a>
<i>HAPI FHIR</i>	<i>The HAPI FHIR library is an implementation of the <a href="#">HL7 FHIR</a> specification for Java.</i>	<a href="#">HAPI FHIR</a>
<i>HL7 FHIR</i>	<i>Fast Healthcare Interoperability Resources (FHIR) are a standard for health care data exchange, published by HL7®.</i>	<a href="#">HL7 FHIR</a>

Term	Definition	Links
<i>Implementation Guide</i>	<i>A coherent and bounded set of adaptations that are published as a single unit. Validation occurs within the context of the Implementation Guide.</i>	<a href="#">Profiling FHIR</a>
<i>JavaPoet</i>	<i>JavaPoet is a Java API for generating <code>.java</code> source files.</i>	<a href="#">JavaPoet (Github)</a>
<i>Mockito</i>	<i>Mockito is a mocking framework that provides a clean &amp; simple API, resulting in readable tests and clean verification errors.</i>	<a href="#">Mockito</a>
<i>Resource</i>	<i>A resource is an entity that has a known identity by which it can be addressed, identifies itself as one of the types of resource defined in the <a href="#">HL7 FHIR</a> specification, contains a set of structured data items as described by the definition of the resource type and has an identified version that changes if the contents of the resource change.</i>	<a href="#">Base Resource Definitions</a>
<i>Snapshot statement</i>	<i>Snapshot statements are one of two ways to describe the inner structure of a <a href="#">StructureDefinition</a>. Snapshot statements are a fully calculated form of the structure that is not dependent on any other structure.</i>	<a href="#">Base Resource Definitions</a>
<i>StructureDefinition</i>	<i>A definition of a FHIR structure. This resource is used to describe the underlying resources, data types defined in FHIR, and also for describing extensions and constraints on resources and data types.</i>	<a href="#">Resource StructureDefinition - Content</a>
<i>ValueSet</i>	<i>A ValueSet resource instance specifies a set of codes drawn from one or more code systems, intended for use in a particular context. ValueSets link between <a href="#">CodeSystems</a> definitions and their use in coded elements.</i>	<a href="#">Resource ValueSet - Content</a>