

Scala 3 User's Manual

Internal Draft (rev cc47c561aa)

Martin Odersky et al.

17 July 2021



Latest HTML version available at <https://dotty.epfl.ch/docs/usage/>

Contents

1	Getting Started: Users	4
1.1	Trying out Scala 3	4
2	Using Dotty with sbt	5
3	IDE support for Dotty	5
3.1	Using Visual Studio Code	5
3.2	Using IntelliJ IDEA	6
4	Worksheet mode with Dotty IDE	6
5	How to use the worksheets	7
6	Implementation details	8
7	Language Versions	8
8	Using Dotty with cbt	9
9	Scaladoc	10
9.1	Other extensions	10
10	API Documentation	10
11	Built-in blog	10
12	{{page.title}}	10
12.1	Proper directory setup	10
12.2	Naming convention	11
13	Scaladoc-specific Tags and Features	11
14	Static documentation	11
15	{{ page.title}}	11
15.1	Properties	11
15.2	Using existing Templates and Layouts	12
15.3	Sidebar	13
15.4	Static resources	13
16	Dottydoc [Legacy]	15
17	Using existing Templates and Layouts	15
18	Blog	16
19	Includes	16

20	Sidebar	17
21	Dottydoc Specific Tags and Behavior	18
21.1	Linking to API	18
21.2	Rendering Docstrings	18
21.3	Other extensions	18
22	Default Layouts	18
22.1	main.html	18
22.2	sidebar.html	18
22.3	doc-page.html	19
22.4	api-page.html	19
22.5	blog-page.html	19
23	Default Includes	19

1 Getting Started: Users [↗](#)

1.1 Trying out Scala 3

1.1.1 In your web browser

[Scastie](#), the online Scala playground, supports Dotty. This is an easy way to try Scala 3 without installing anything, directly in your browser.

1.1.2 sbt

The fastest way to create a new project compiled by Scala 3 is using [sbt](#).

Create a simple Scala 3 project:

```
$ sbt new scala/scala3.g8
```

Or a Scala 3 project that cross compiles with Scala 2:

```
$ sbt new scala/scala3-cross.g8
```

You can then start a Scala 3 REPL directly from your sbt project:

```
$ sbt
> console
scala>
```

For more information, see the [Scala 3 Example Project](#).

1.1.3 IDE support

Start using the Dotty IDE in any Dotty project by following the [IDE guide](#).

1.1.4 Standalone installation

Releases are available for download on the [Releases Section](#) of the Dotty repository. Releases include three executables: - scalac, the Scala 3 compiler, - scaladoc, the [Scaladoc](#) and - scala, the Scala 3 REPL.

```
.
└─ bin
    ├── scalac
    ├── scaladoc
    └── scala
```

Add these executables to your PATH and you will be able to run the corresponding commands directly from your console:

```
# Compile code using Scala 3
$ scalac HelloWorld.scala
```

```
# Run it with the proper classpath
```

```
$ scala HelloWorld
```

```
# Start a Scala 3 REPL
```

```
$ scala
```

```
Starting scala3 REPL...
```

```
scala>
```

If you're a Mac user, we also provide a [homebrew](#) package that can be installed by running:

```
brew install lampepfl/brew/dotty
```

In case you have already installed Dotty via brew, you should instead update it:

```
brew upgrade dotty
```

1.1.5 Scala 3 for Scripting

If you have followed the steps in “Standalone Installation” section and have the `scala` executable on your [PATH](#), you can run `*.scala` files as scripts. Given a source file named `Test.scala`:

```
@main def Test(name: String): Unit =  
  println(s"Hello ${name}!")
```

You can run: `scala Test.scala World` to get the output `Hello World!`.

A “script” is an ordinary Scala file which contains a main method. The semantics of the `scala Script.scala` command is as follows:

- Compile `Script.scala` with `scalac` into a temporary directory.
- Detect the main method in the `*.class` files produced by the compilation.
- Execute the main method.

2 Using Dotty with sbt

To try it in your project see the [Getting Started User Guide](#).

3 IDE support for Dotty

IDE support for Scala 3 is available in IDEs based on [Scala Metals](#) (e.g., Visual Studio Code, vim) and in [IntelliJ IDEA](#).

3.1 Using Visual Studio Code

To use Visual Studio Code on a Scala 3 project, ensure you have the [Metals](#) plugin installed. Then open the project directory in VS code and click the “Import build” button in notification.

3.1.1 Under the Hood

VS Code implements semantic features (such as completions, “go to definition”) using the [Language Server Protocol \(LSP\)](#), so it needs a language server implementation. Metals is the implementation of LSP for Scala. It extracts semantic information from [semanticdb](#), which is generated directly by the Scala 3 compiler.

You can read more about Scala 3 support in Metals in [this blog post](#).

To communicate with the build tool (e.g., to import the project, trigger builds, run tests), Metals uses the [Build Server Protocol \(BSP\)](#). The default BSP implementation used by metals is [Bloop](#), which supports Scala 3 projects. Alternatively, [sbt can be used as a BSP server](#) as it directly implements BSP since version 1.4.

3.2 Using IntelliJ IDEA

IntelliJ has its own implementation for semantic features, so it does not use Metals or the Language Server Protocol (LSP).

In order to import a project into IntelliJ there are two possibilities:

- Use the built-in feature to import sbt builds
- Use IntelliJ’s support for the [Build Server Protocol \(BSP\)](#)

3.2.1 Importing the sbt build

To use IntelliJ’s sbt import, go to “File” - “Open...” and select your project’s `build.sbt` file.

In this mode, IntelliJ starts sbt with a custom plugin to extract the project structure. After importing, IntelliJ no longer interacts with other sbt sessions. Building and running the project within the IDE is done by separate processes.

3.2.2 Importing the project using BSP

To import a project using BSP, go to “File” - “New” - “Project from Existing Sources” and select the project directory. In the upcoming dialog select “BSP” to import the project. You may be asked to choose between “sbt” and “sbt with Bloop”, the recommended option is “sbt”.

If the project import fails (“Problem executing BSP job”), navigate to your project in a terminal and just start sbt. Once sbt is running, open the “bsp” tab in IntelliJ click the “Reload” button.

When using IntelliJ’s BSP mode, build and run commands from the IDE are executed through sbt, so they have the same effect as building or running the project through sbt in the terminal.

4 Worksheet mode with Dotty IDE

A worksheet is a Scala file that is evaluated on save, and the result of each expression is shown in a column to the right of your program. Worksheets are like a REPL session on steroids,

and enjoy 1st class editor support: completion, hyperlinking, interactive errors-as-you-type, etc. Worksheet use the extension `.sc`.

5 How to use the worksheets

The only supported client for the Worksheet mode is [Visual Studio Code](#).

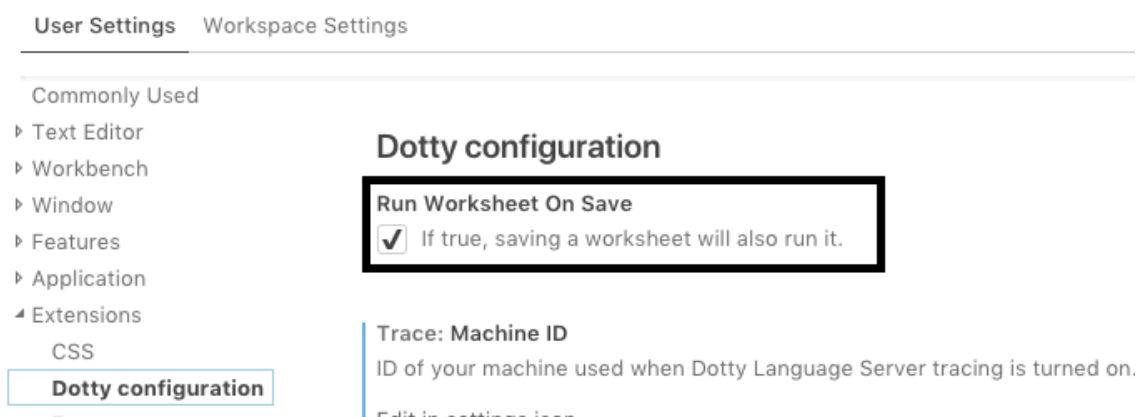
To use the worksheets, start Dotty IDE by [following the instruction](#) and create a new file `MyWorksheet.sc` and write some code:

```
val xyz = 123
println("Hello, worksheets!")
456 + xyz
```

On top of the buffer, the message `Run this worksheet` appears. Click it to evaluate the code of the worksheet. Each line of output is printed on the right of the expression that produced it. The worksheets run with the classes of your project and its dependencies on their classpath.

	Run this worksheet	
1	<code>val foo = 123</code>	<code>val foo: Int = 123</code>
2	<code>def bar(x: Int) = x + 1</code>	<code>def bar(x: Int): Int</code>
3	<code>object 0 { val fizz = foo + 3 }</code>	<code>// defined object 0</code>
4	<code>bar(0.fizz + foo)</code>	<code>val res0: Int = 250</code>

By default, the worksheets are also run when the file is saved. This can be configured in VSCode preferences:



Note that the worksheet are fully integrated with the rest of Dotty IDE: While typing, errors are shown, completions are suggested, and you can use all the other features of Dotty IDE such as go to definition, find all references, etc.

```

Run this worksheet
1  val foo = 123
2  def bar(x: Int) = x + 1
3  object bar(x: Int): Int { foo + 3 }
4  bar()

```

6 Implementation details

The implementation details of the worksheet mode and the information necessary to add support for other clients are available in [Worksheet mode - Implementation details](#).

7 Language Versions [↗](#)

The default Scala language version currently supported by the Dotty compiler is 3.0. There are also other language versions that can be specified instead:

- 3.0-migration: Same as 3.0 but with a Scala 2 compatibility mode that helps moving Scala 2.13 sources over to Scala 3. In particular, it
 - flags some Scala 2 constructs that are disallowed in Scala 3 as migration warnings instead of hard errors,
 - changes some rules to be more lenient and backwards compatible with Scala 2.13
 - gives some additional warnings where the semantics has changed between Scala 2.13 and 3.0
 - in conjunction with `-rewrite`, offer code rewrites from Scala 2.13 to 3.0.
- future: A preview of changes introduced in the next versions after 3.0. In the doc pages here we refer to the language version with these changes as 3.1, but it might be that some of these changes will be rolled out in later 3.x versions.

Some Scala-2 specific idioms will be dropped in this version. The feature set supported by this version will be refined over time as we approach its release.

- future-migration: Same as future but with additional helpers to migrate from 3.0. Similarly to the helpers available under 3.0-migration, these include migration warnings and optional rewrites.

There are two ways to specify a language version.

- With a `-source` command line setting, e.g. `-source 3.0-migration`.
- With a `scala.language` import at the top of a source file, e.g:

```

package p
import scala.language.`future-migration`

class C { ... }

```


Language imports supersede command-line settings in the source files where they are specified. Only one language import specifying a source version is allowed in a source file, and it must come before any definitions in that file.

8 Using Dotty with cbt

NOTE: cbt support for Scala 3 is experimental and incomplete (for example, incremental compilation is not supported), we recommend [using Scala 3 with sbt](#) for now.

cbt comes with built-in Scala 3 support. Follow the [cbt tutorial](#), then simply extend Dotty in the Build class.

```
// build/build.scala
import cbt.*
class Build(val context: Context) extends Dotty {
  ...
}
```

Also see the [example project](#).

9 Scaladoc

scaladoc logo

scaladoc is a tool to generate documentation for your Scala 3 projects. It provides similar features to javadoc as well as jekyll or docusaurus.

As you probably have guessed, this whole site was created using scaladoc.

```
{% for post in site.posts %} ## {{ post.title }}
```

```
{{ post.excerpt }}
```

[\(read more\)](#)

```
{% endfor %}
```

9.1 Other extensions

We would love to have your feedback on what you think would be good in order to render the documentation you want! Perhaps you would like to render method definitions or members? Do you want to have runnable code snippets? Let us know by filing [issues](#)!

10 API Documentation

11 Built-in blog

12 {{page.title}}

Scaladoc allows you to include a simple blog in your documentation. For now, it provides only basic features. In the future, we plan to include more advanced features like tagging or author pages.

Blog is treated a little differently than regular static sites. This article will help you set up your own blog.

12.1 Proper directory setup

All your blogposts must be put under blog/_posts directory.

```
├─ blog
│   └─ _posts
│       └─ 2016-12-05-implicit-function-types.md
│       └─ index.html
```

If you are using yaml [sidebar](#) don't forget to place

```
sidebar:
  - title: Blog
```

somewhere inside the `yaml` tree representing the sidebar sections. Scaladoc will attach under that section all of your blogposts.

12.2 Naming convention

All the blogpost filenames should start with date in numeric format matching `YYYY-MM-DD`. Example name is `2015-10-23-dotty-compiler-bootstrap.md`.

13 Scaladoc-specific Tags and Features [↗](#)

14 Static documentation [↗](#)

15 {{ page.title }}

Scaladoc is able to generate static sites, known from [Jekyll](#) or [Docusaurus](#). Having a combined tool allows to provide interaction between static documentation and API, thus allowing the two to blend naturally.

Creating a site is just as simple as in Jekyll. The site root contains the layout of the site and all files placed there will be either considered static, or processed for template expansion.

The files that are considered for template expansion must end in `*.{html,md}` and will from here on be referred to as “template files” or “templates”.

A simple “hello world” site could look something like this:

```
├─ docs
│   └─ getting-started.md
└─ index.html
```

This will give you a site with the following files in generated documentation:

```
index.html
docs/getting-started.html
```

Scaladoc can transform both files and directories (to organize your documentation into tree-like structure). By default directories has title based on file name and has empty content. There is an option to include `index.html` or `index.md` (not both) to provide both content and properties like title (see [Properties](#)).

15.1 Properties

Scaladoc uses the [Liquid](#) templating engine and provides a number of custom filters and tags specific to Scala documentation.

In Scaladoc, all templates can contain YAML front-matter. The front-matter is parsed and put into the `page` variable available in templates via Liquid.

Scaladoc uses some predefined properties to controls some aspect of page.

Predefined properties:

- title provide page title that will be used in navigation and html metadata.
- extraCss additional .css files that will be included in this page. Paths should be relative to documentation root. This setting is not exported to template engine.
- extraJs additional .js files that will be included in this page. Paths should be relative to documentation root. This setting is not exported to template engine.
- hasFrame when set to false page will not include default layout (navigation, breadcrumbs etc.) but only token html wrapper to provide metadata and resources (js and css files). This setting is not exported to template engine.
- layout - predefined layout to use, see below. This setting is not exported to template engine.

15.2 Using existing Templates and Layouts

To perform template expansion, Dottydoc looks at the layout field in the front-matter. Here's a simple example of the templating system in action, index.html:

```
---
layout: main
---

<h1>Hello world!</h1>
```

With a simple main template like this:

```
{% raw %}
<html>
  <head>
    <title>Hello, world!</title>
  </head>
  <body>
    {{ content }}
  </body>
</html>
```

Would result in {{ content }} being replaced by <h1>Hello world!</h1> from the index.html file. {% endraw %}

Layouts must be placed in a _layouts directory in the site root:

```
|— _layouts
|   |— main.html
|— docs
|   |— getting-started.md
|— index.html
```

15.3 Sidebar

Scaladoc by default uses layout of files in docs directory to create table of content. There is also ability to override it by providing a `sidebar.yml` file in the site root:

```
sidebar:
  - title: Blog
  - title: Docs
    url: docs/index.html
  - title: Usage
    subsection:
      - title: Dottydoc
        url: docs/usage/dottydoc.html
      - title: sbt-projects
        url: docs/usage/sbt-projects.html
```

The sidebar key is mandatory, as well as title for each element. The default table of contents allows you to have subsections - albeit the current depth limit is 2 however it accepts both files and directories and latter can be used to provide deeper structures.

The items must provide either subsection or url but not both at once! The only exception is Blog which is only a title and behaves differently. You can read more about blog [here](#).

```
|— blog
|   |— _posts
|   |   └─ 2016-12-05-implicit-function-types.md
|   └─ index.html
|— index.html
└─ sidebar.yml
```

15.4 Static resources

You can attach static resources (pdf, images) to your documentation by using two dedicated directories: `resources` and `images`. When you upload your assets under any of these directories you can reference them in markdown as if they were relatively at the same level.

For example, consider the following situation:

```
|— blog
|   |— _posts
|   |   └─ 2016-12-05-implicit-function-types.md
|   └─ index.html
|— index.html
|— resources
|   └─ my_file.pdf
|— images
|   └─ my_image.png
└─ sidebar.yml
```

You can refer to the assets from within any of the files using markdown links:

This is my blogpost. Here is the image and here is my [pdf](my_file.

16 Dottydoc [Legacy]

Dottydoc is a tool to generate a combined documentation and API reference for your project.

In previous versions of the Scaladoc tool, there is a big divide between what is documentation and what is API reference. Dottydoc allows referencing, citing and rendering parts of your API in your documentation, thus allowing the two to blend naturally.

To do this, Dottydoc is very similar to what [Jekyll](#) provides in form of static site generation. As you probably guessed, this whole site was created using Dottydoc.

Creating a site is just as simple as in Jekyll. The site root contains the layout of the site and all files placed here will be either considered static, or processed for template expansion.

The files that are considered for template expansion must end in `*.{html,md}` and will from here on be referred to as “template files” or “templates”.

A simple “hello world” site could look something like this:

```
├─ docs
│   └─ getting-started.md
└─ index.html
```

This will give you a site with the following endpoints:

```
_site/index.html
_site/docs/getting-started.html
```

Just as with Jekyll, the site is rendered in a `_site` directory.

17 Using existing Templates and Layouts

Dottydoc uses the [Liquid](#) templating engine and provides a number of custom filters and tags specific to Scala documentation.

In Dottydoc, all templates can contain YAML front-matter. The front-matter is parsed and put into the page variable available in templates via Liquid.

To perform template expansion, Dottydoc looks at `layout` in the front-matter. Here’s a simple example of the templating system in action, `index.html`:

```
---
layout: main
---

<h1>Hello world!</h1>
```

With a simple main template like this:

```
{% raw %}

<html>
  <head>
```

```

    <title>Hello, world!</title>
  </head>
  <body>
    {{ content }}
  </body>
</html>

```

Would result in `{{ content }}` being replaced by `<h1>Hello world!</h1>` from the `index.html` file. `{% endraw %}`

Layouts must be placed in a `_layouts` directory in the site root:

```

├─ _layouts
│   └─ main.html
├─ docs
│   └─ getting-started.md
└─ index.html

```

It is also possible to use one of the **default layouts** that ship with Dottydoc.

18 Blog

Dottydoc also allows for a simple blogging platform in the same vein as Jekyll. Blog posts are placed within the `./blog/_posts` directory and have to be in the form `year-month-day-title.{md,html}`.

An example of this would be:

```

├─ blog
│   └─ _posts
│       └─ 2016-12-05-implicit-function-types.md
└─ index.html

```

To be rendered as templates, each blog post should have front-matter and a layout declaration.

The posts are also available in the variable `site.posts` throughout the site. The fields of these objects are the same as in `[BlogPost]` (`dotty.tools.dottydoc.staticsite.BlogPost`).

19 Includes

In Liquid, there is a concept of include tags, these are used in templates to include other de facto templates:

```

<div class="container">
  {% raw %}{% include "sidebar.html" %}{% endraw %}
</div>

```

You can leave out the file extension if your include ends in `.html`.

Includes need to be kept in `_includes` in the site root. Dottydoc provides a couple of **default includes**, but the user-specified includes may override these.

An example structure with an include file “`sidebar.html`”:

```

├── _includes
│   └── sidebar.html
├── blog
│   ├── _posts
│   │   └── 2016-12-05-implicit-function-types.md
│   └── index.md
└── index.html

```

20 Sidebar

Dottydoc gives you the ability to create your own custom table of contents, this can either be achieved by overriding the `toc.html` or by providing a `sidebar.yml` file in the site root:

```

sidebar:
  - title: Blog
    url: blog/index.html
  - title: Docs
    url: docs/index.html
  - title: Usage
    subsection:
      - title: Dottydoc
        url: docs/usage/dottydoc.html
      - title: sbt-projects
        url: docs/usage/sbt-projects.html

```

The `sidebar` key is mandatory, as well as `title` for each element. The default table of contents allows you to have subsections - albeit the current depth limit is 2 – we would love to see this changed, contributions welcome!

The items which have the `subsection` key, may not have a `url` key in the current scheme. A site root example with this could be:

```

├── blog
│   └── _posts
│       └── 2016-12-05-implicit-function-types.md
├── index.html
└── sidebar.yml

```

21 Dottydoc Specific Tags and Behavior

21.1 Linking to API

If you for instance, want to link to `scala.collection.immutable.Seq` in a markdown file, you can simply use the canonical path in your url:

```
[Seq](scala.collection.immutable.Seq)
```

Linking to members is done in the same fashion:

```
[Seq](scala.collection.immutable.Seq.isEmpty)
```

Dottydoc denotes objects by ending their names in “\$”. To select `List.range` you’d therefore write:

```
[List.range](scala.collection.immutable.List$.range)
```

21.2 Rendering Docstrings

Sometimes you end up duplicating the docstring text in your documentation, therefore Dottydoc makes it easy to render this inline:

```
{% raw %}{% docstring "scala.collection.immutable.Seq" %}{% endraw %}
```

21.3 Other extensions

We would love to have your feedback on what you think would be good in order to render the documentation you want! Perhaps you would like to render method definitions or members? Let us know by filing [issues](#)!

22 Default Layouts

22.1 main.html

A wrapper for all other layouts, includes a default `<head>` with included JavaScripts and CSS style-sheets.

22.1.1 Variables

- `content`: placed in `<body>` tag
- `extraCSS`: a list of relative paths to extra CSS style-sheets for the site
- `extraJS`: a list of relative paths to extra JavaScripts for the site
- `title`: the `<title>` of the page

22.2 sidebar.html

Sidebar uses `main.html` as its parent layout. It adds a sidebar generated from a YAML file (if exists), as well as the index for the project API.

22.2.1 Variables

- `content`: placed in a `<div>` with class `content-body`
- `docs`: the API docs generated from supplied source files, this is included by default and does not need to be specified.

22.3 doc-page.html

Doc page is used for pages that need a sidebar and provides a small wrapper for the included `{% raw %}{{ content }}{% endraw %}`.

22.4 api-page.html

The last two layouts are special, in that they are treated specially by Dottydoc. The input to the API page is a documented `[Entity](dotty.tools.dottydoc.model.Entity)`. As such, this page can be changed to alter the way Dottydoc renders API documentation.

22.5 blog-page.html

A blog page uses files placed in `./blog/_posts/` as input to render a blog.

23 Default Includes

- `scala-logo.svg`: the scala in Dotty version as svg
- `toc.html`: the default table of contents template