

Spring Initializr Reference Guide

Stéphane Nicoll , Dave Syer

Copyright ©

Table of Contents

1. Spring Initializr Documentation	1
1.1. About the documentation	1
1.2. Getting help	1
2. Metadata Format	2
2.1. Content	2
Project dependencies	6
Project types	7
Packaging	7
Java version	8
Languages	8
Boot version	8
2.2. Defaults	8
3. Using the Stubs	9
3.1. Using WireMock with Spring Boot	9
3.2. Names and Paths of Stubs	10
4. Configuration Format	11
4.1. Env section	11
4.2. Dependencies section	12
Dependency group	13
4.3. Other sections	13

1. Spring Initializr Documentation

Spring Initializr provides an extensible API to generate quickstart projects. It also provides a configurable service: you can see our default instance at start.spring.io. It provides a simple web UI to configure the project to generate and endpoints that you can use via plain HTTP.

Spring Initializr also exposes an endpoint that serves its metadata in a [well-known format](#) to allow third-party clients to provide the necessary assistance.

Finally, Initializr offers a [configuration structure](#) to define all the aspects related to the project to generate: list of dependencies, supported java and boot versions, etc.

1.1 About the documentation

The Spring Initializr reference guide is available as [html](#), [pdf](#) and [epub](#) documents. The latest copy is available at docs.spring.io/initializr/docs/current/reference.

Copies of this document may be made for your own use and for distribution to others, provided that you do not charge any fee for such copies and further provided that each copy contains this Copyright Notice, whether distributed in print or electronically.

1.2 Getting help

Having trouble with Spring Initializr, We'd like to help!

- Ask a question on [Gitter](#).
- Report bugs with Spring Initializr at github.com/spring-io/initializr/issues.

Note

All of Spring Initializr is open source, including the documentation! If you find problems with the docs; or if you just want to improve them, please [get involved](#).

2. Metadata Format

This section describes the hal/json structure of the metadata exposed by the initializr. Such metadata can be used by third party clients to provide a list of options and default settings that can be used to request the creation of a project.

Each third-party client is advised to set a `User-Agent` header for **each** request sent to the service. A good structure for a user agent is `clientId/clientVersion` (i.e. `foo/1.2.0` for the "foo" client and version `1.2.0`).

2.1 Content

Any third party client can retrieve the capabilities of the service by issuing a `GET` on the root URL using the following `Accept` header: `application/vnd.initializr.v2.1+json`. Please note that the metadata may evolve in a non backward compatible way in the future so adding this header ensures the service returns the metadata format you expect.

This is an example output for a service running at start.spring.io:

request.

```
GET / HTTP/1.1
Accept: application/vnd.initializr.v2.1+json
Host: start.spring.io
```

response.

```
HTTP/1.1 200 OK
ETag: "1262473f5c28f970e1e42d107213b064"
Content-Type: application/vnd.initializr.v2.1+json
Cache-Control: max-age=604800
Content-Length: 4872

{
  "_links" : {
    "maven-build" : {
      "href" : "https://start.spring.io/pom.xml?type=maven-
build{&dependencies,packaging,javaVersion,language,bootVersion,groupId,artifactId,version,name,description,packageName}",
      "templated" : true
    },
    "maven-project" : {
      "href" : "https://start.spring.io/starter.zip?type=maven-
project{&dependencies,packaging,javaVersion,language,bootVersion,groupId,artifactId,version,name,description,packageName}",
      "templated" : true
    },
    "gradle-build" : {
      "href" : "https://start.spring.io/build.gradle?type=gradle-
build{&dependencies,packaging,javaVersion,language,bootVersion,groupId,artifactId,version,name,description,packageName}",
      "templated" : true
    },
    "gradle-project" : {
      "href" : "https://start.spring.io/starter.zip?type=gradle-
project{&dependencies,packaging,javaVersion,language,bootVersion,groupId,artifactId,version,name,description,packageName}",
      "templated" : true
    },
    "dependencies" : {
      "href" : "https://start.spring.io/dependencies{?bootVersion}",
      "templated" : true
    }
  },
  "dependencies" : {
    "type" : "hierarchical-multi-select",
    "values" : [ {
```

```

    "name" : "Core",
    "values" : [ {
      "id" : "web",
      "name" : "Web",
      "description" : "Web dependency description",
      "_links" : {
        "guide" : {
          "href" : "https://example.com/guide",
          "title" : "Building a RESTful Web Service"
        },
        "reference" : {
          "href" : "https://example.com/doc"
        }
      }
    }, {
      "id" : "security",
      "name" : "Security"
    }, {
      "id" : "data-jpa",
      "name" : "Data JPA"
    } ]
  }, {
    "name" : "Other",
    "values" : [ {
      "id" : "org.acme:foo",
      "name" : "Foo",
      "_links" : {
        "guide" : [ {
          "href" : "https://example.com/guide1"
        }, {
          "href" : "https://example.com/guide2",
          "title" : "Some guide for foo"
        } ],
        "reference" : {
          "href" : "https://example.com/{bootVersion}/doc",
          "templated" : true
        }
      }
    }, {
      "id" : "org.acme:bar",
      "name" : "Bar"
    }, {
      "id" : "org.acme:biz",
      "name" : "Biz",
      "versionRange" : "1.2.0.BUILD-SNAPSHOT"
    }, {
      "id" : "org.acme:bur",
      "name" : "Bur",
      "versionRange" : "[1.1.4.RELEASE,1.2.0.BUILD-SNAPSHOT)"
    }, {
      "id" : "my-api",
      "name" : "My API"
    } ]
  } ]
},
"type" : {
  "type" : "action",
  "default" : "maven-project",
  "values" : [ {
    "id" : "maven-build",
    "name" : "Maven POM",
    "action" : "/pom.xml",
    "tags" : {
      "build" : "maven",
      "format" : "build"
    }
  }, {
    "id" : "maven-project",
    "name" : "Maven Project",
    "action" : "/starter.zip",
    "tags" : {

```

```

        "build" : "maven",
        "format" : "project"
    }
}, {
    "id" : "gradle-build",
    "name" : "Gradle Config",
    "action" : "/build.gradle",
    "tags" : {
        "build" : "gradle",
        "format" : "build"
    }
}, {
    "id" : "gradle-project",
    "name" : "Gradle Project",
    "action" : "/starter.zip",
    "tags" : {
        "build" : "gradle",
        "format" : "project"
    }
} ]
},
"packaging" : {
    "type" : "single-select",
    "default" : "jar",
    "values" : [ {
        "id" : "jar",
        "name" : "Jar"
    }, {
        "id" : "war",
        "name" : "War"
    } ]
},
"javaVersion" : {
    "type" : "single-select",
    "default" : "1.8",
    "values" : [ {
        "id" : "1.6",
        "name" : "1.6"
    }, {
        "id" : "1.7",
        "name" : "1.7"
    }, {
        "id" : "1.8",
        "name" : "1.8"
    } ]
},
"language" : {
    "type" : "single-select",
    "default" : "java",
    "values" : [ {
        "id" : "groovy",
        "name" : "Groovy"
    }, {
        "id" : "java",
        "name" : "Java"
    }, {
        "id" : "kotlin",
        "name" : "Kotlin"
    } ]
},
"bootVersion" : {
    "type" : "single-select",
    "default" : "1.1.4.RELEASE",
    "values" : [ {
        "id" : "1.2.0.BUILD-SNAPSHOT",
        "name" : "Latest SNAPSHOT"
    }, {
        "id" : "1.1.4.RELEASE",
        "name" : "1.1.4"
    }, {
        "id" : "1.0.2.RELEASE",

```

```

    "name" : "1.0.2"
  } ]
},
"groupId" : {
  "type" : "text",
  "default" : "com.example"
},
"artifactId" : {
  "type" : "text",
  "default" : "demo"
},
"version" : {
  "type" : "text",
  "default" : "0.0.1-SNAPSHOT"
},
"name" : {
  "type" : "text",
  "default" : "demo"
},
"description" : {
  "type" : "text",
  "default" : "Demo project for Spring Boot"
},
"packageName" : {
  "type" : "text",
  "default" : "com.example"
}
}

```

The current capabilities are the following:

- Project dependencies: these are the *starters* really or actually any dependency that we might want to add to the project.
- Project types: these define the action that can be invoked on this service and a description of what it would produce (for instance a zip holding a pre-configured Maven project). Each type may have one more tags that further define what it generates.
- Packaging: the kind of projects to generate. This merely gives a hint to the component responsible to generate the project (for instance, generate an executable *jar* project).
- Java version: the supported java versions
- Language: the language to use (e.g. Java)
- Boot version: the Spring Boot version to use
- Additional basic information such as: `groupId`, `artifactId`, `version`, `name`, `description` and `packageName`.

Each top-level attribute (i.e. capability) has a standard format:

- A `type` attribute that defines the semantic of the attribute (see below).
- A `default` attribute that defines either the default value or the reference to the default value.
- A `values` attribute that defines the set of acceptable values (if any). This can be hierarchical (with `values` being held in `values`). Each item in a `values` array can have an `id`, `name` and `description`).

The following attribute `type` are supported:

- `text`: defines a simple text value with no option.

- `single-select`: defines a simple value to be chosen amongst the specified options.
- `hierarchical-multi-select`: defines a hierarchical set of values (values in values) with the ability to select multiple values.
- `action`: a special type that defines the attribute defining the action to use.

Each action is defined as a HAL-compliant URL. For instance, the `maven-project` type templated URL is defined as follows:

Type link example.

```
{
  "href" : "https://start.spring.io/starter.zip?type=maven-
project{%dependencies,packaging,javaVersion,language,bootVersion,groupId,artifactId,version,name,description,packageName}",
  "templated" : true
}
```

You can use Spring HATEOAS and the `UriTemplate` helper in particular to generate an URI from template variables. Note that the variables match the name of top-level attribute in the metadata document. If you can't parse such URI, the `action` attribute of each type gives you the root action to invoke on the server. This requires more manual handling on your end.

Project dependencies

A dependency is usually the coordinates of a *starter* module but it can be just as well be a regular dependency. A typical dependency structure looks like this:

```
{
  "name": "Display name",
  "id": "org.acme.project:project-starter-foo",
  "description": "What starter foo does"
}
```

The name is used as a display name to be shown in whatever UI used by the remote client. The id can be anything, really as the actual dependency definition is defined through configuration. If no id is defined, a default one is built using the `groupId` and `artifactId` of the dependency. Note in particular that the version is **never** used as part of an automatic id.

Each dependency belongs to a group. The idea of the group is to gather similar dependencies and order them. Here is a value containing the `core` group to illustrates the feature:

Dependency group example.

```
{
  "name" : "Core",
  "values" : [ {
    "id" : "web",
    "name" : "Web",
    "description" : "Web dependency description",
    "_links" : {
      "guide" : {
        "href" : "https://example.com/guide",
        "title" : "Building a RESTful Web Service"
      },
      "reference" : {
        "href" : "https://example.com/doc"
      }
    }
  }, {
    "id" : "security",
```



```

    "name" : "Security"
  }, {
    "id" : "data-jpa",
    "name" : "Data JPA"
  } ]
}

```

Each dependency can have *links* (in a HAL-compliant format). Links are grouped by "relations" that provide a semantic to the link. A link can also have a *title* and its URI can be templated. At the moment, the only valid parameter is `bootVersion`.

The official relations are:

- `guide`: link to an how-to or guide that explain how to get started
- `reference`: link to a section of a reference guide (documentation)

Project types

The `type` element defines what kind of project can be generated and how. For instance, if the service exposes the capability to generate a Maven project, this would look like this:

Project type example.

```

{
  "id" : "maven-build",
  "name" : "Maven POM",
  "action" : "/pom.xml",
  "tags" : {
    "build" : "maven",
    "format" : "build"
  }
}

```

You should not rely on the output format depending that information. Always use the response headers that define a `Content-Type` and also a `Content-Disposition` header.

Note that each `id` has a related HAL-compliant link that can be used to generate a proper URI based on template variables. The top-level `type` has, as any other attribute, a `default` attribute that is a hint to select what the service consider to be a good default.

The `action` attribute defines the endpoint the client should contact to actually generate a project of that type if you can't use the HAL-compliant url.

The `tags` object is used to categorize the project type and give *hints* to 3rd party client. For instance, the *build* tag defines the build system the project is going to use and the *format* tag defines the format of the generated content (i.e. here a complete project vs. a build file. Note that the `Content-type` header of the reply provides additional metadata).

Packaging

The `packaging` element defines the kind of project that should be generated.

Packaging example.

```

{
  "id" : "jar",
  "name" : "Jar"
}

```

The obvious values for this element are `jar` and `war`.

Java version

The `javaVersion` element provides a list of possible java versions for the project:

Java example.

```
{
  "id" : "1.6",
  "name" : "1.6"
}
```

Languages

The `language` element provides a list of possible languages for the project:

Language example.

```
{
  "id" : "groovy",
  "name" : "Groovy"
}
```

Boot version

The `bootVersion` element provides the list of available boot versions

Spring Boot version example.

```
{
  "id" : "1.2.0.BUILD-SNAPSHOT",
  "name" : "Latest SNAPSHOT"
}
```

2.2 Defaults

Each top-level element has a `default` attribute that should be used as a hint to provide the default value in the relevant UI component.

3. Using the Stubs

The Initializr project publishes [WireMock](#) stubs for all the JSON responses that are tested in the project. If you are writing a client for the Initializr service, you can use these stubs to test your own code. You can consume them with the raw Wiremock APIs, or via some features of [Spring Cloud Contract](#).

WireMock is an embedded web server that analyses incoming requests and chooses stub responses based on matching some rules (e.g. a specific header value). So if you send it a request which matches one of its stubs, it will send you a response as if it was a real Initializr service, and you can use that to do full stack integration testing of your client.

3.1 Using WireMock with Spring Boot

A convenient way to consume the stubs in your project is to add a test dependency:

```
<dependency>
  <groupId>io.spring.initializr</groupId>
  <artifactId>initializr-web</artifactId>
  <classifier>stubs</classifier>
  <version>{project-version}</version>
  <scope>test</scope>
</dependency>
```

and then pull the stubs from the classpath. In a Spring Boot application, using Spring Cloud Contract, you can start a WireMock server and register all the stubs with it like this:

```
@RunWith(SpringRunner.class)
@SpringBootTest
@AutoConfigureWireMock(port = 0,
    stubs="classpath:META-INF/io.spring.initializr/initializr-web/0.3.0.BUILD-SNAPSHOT")
public class ClientApplicationTests {

    @Value("${wiremock.server.port}")
    private int port;

    ...

}
```

Alternatively you can configure the stub runner to look for the artifact. The example below will automatically download, if necessary, the latest version of the initializr stubs:

```
@RunWith(SpringRunner.class)
@SpringBootTest(webEnvironment = WebEnvironment.NONE)
@AutoConfigureStubRunner(
    ids = "io.spring.initializr:initializr-web",
    workOffline = true)
public class InitializrIntegrationTests {

    @Autowired
    private StubFinder stubFinder;

    @Autowired
    private RestTemplate restTemplate;

    @Test
    public void testCurrentMetadata() throws IOException {
        RequestEntity<Void> request = RequestEntity.get(createUri("/"))
            .accept(MediaType.valueOf("application/vnd.initializr.v2.1+json"))
            .build();

        ResponseEntity<String> response = this.restTemplate
```

```

        .exchange(request, String.class);
        assertThat(response.getStatusCode()).isEqualTo(HttpStatus.OK);
        // other assertions here
    }

    private URI createUri(String path) {
        String url = this.stubFinder.findStubUrl("initializr-web").toString();
        return URI.create(url + path);
    }

    @TestConfiguration
    static class Config {

        @Bean
        public RestTemplate restTemplate(RestTemplateBuilder builder) {
            return builder.build();
        }

    }
}

```

Tip

If you want to test a specific version or validate your API against multiple versions you can define the version to use in the annotation, something like

```

@AutoConfigureStubRunner(
    ids = "io.spring.initializr:initializr-web:0.3.0.BUILD-SNAPSHOT",
    workOffline = true)
public class InitializrIntegrationTests {
    ...
}

```

Then you have a server that returns the stub of the JSON metadata (metadataWithCurrentAcceptHeader.json) when you send it a header `Accept:application/vnd.initializr.v2.1+json` (as recommended).

3.2 Names and Paths of Stubs

The stubs are laid out in a jar file in a form (under `/mappings`) that can be consumed by WireMock just by setting its file source. The names of the individual stubs are the same as the method names of the test cases that generated them in the Initializr project. So for example there is a test case `"metadataWithV2AcceptHeader"` in `MainControllerIntegrationTests` that makes assertions about the response when the accept header is `application/vnd.initializr.v2.1+json`. The response is recorded in the stub, and it will match in WireMock if the same headers and request parameters that were used in the Initializr test case and used in the client. The method name usually summarizes what those values are.

4. Configuration Format

This section describes the configuration structure that is used by the initializr. The metadata provided through configuration are driving the options exposed by a particular instance and [the project metadata format](#).

Tip

A good way to get started with the configuration is to look at the [configuration of the production instance](#) and check the end-result on [start.spring.io](#)

The configuration is split in several sections:

- An `env` section used to provide various global settings.
- A `dependencies` section lists the available dependencies. This is the most important section of the service as it defines the "libraries" that the user can choose.
- The `groupId`, `artifactId`, `version`, `name`, `description` and `packageName` provide default values for these project settings.
- The `types`, `packagings`, `javaVersions`, `languages` and `bootVersions` provide the list of available option for each setting and which one is the default.

4.1 Env section

Tip

Check [the code](#) for a full list of the available configuration options.

The `env` element defines environment option that the service uses:

- `artifactRepository`: the URL of the (maven) repository that should be used to download the Spring Boot CLI distribution bundle. This is only used by the `/spring` endpoint at the moment.
- `springBootMetadataUrl` the URL of the resource that provides the list of available Spring Boot versions..
- `forceSsl`: a boolean flag that determines if we should use `https` even when browsing a resource via `http`. This is *enabled* by default.
- `fallbackApplicationName`: the name of the *default* application. Application names are generated based on the project's name. However, some user input may result in an invalid identifier for a Java class name for instance.
- `invalidApplicationNames`: a fixed list of invalid application names. If a project generation uses one of these names, the fallback is used instead.
- `invalidPackageNames`: a fixed list of invalid package names. If a project generation uses one of these names, the default is used instead.
- `googleAnalyticsTrackingCode`: the Google Analytics code to use. If this is set, Google analytics is automatically enabled.

- `kotlin`: kotlin-specific settings. For now, only the kotlin version to use can be configured.
- `maven`: maven-specified settings. A custom maven parent POM can be defined and whether or not the `spring-boot-dependencies` BOM should be automatically added to the project.

If some of your dependencies require a custom Bill of Materials (BOM) and/or a custom repository, you can add them here and use the id as a reference. For instance, let's say that you want to integrate with library `foo` and it requires a `foo-bom` and a `foo-repo`. You can configure things as follows:

```
initializr:
  env:
    boms:
      foo-bom:
        groupId: com.example
        artifactId: foo-bom
        version: 1.2.3
    repositories:
      foo-repo:
        name: foo-release-repo
        url: https://repo.example.com/foo
        snapshotsEnabled: false
```

You can then use the `foo-bom` and `foo-repo` in a "dependency" or "dependency group" section.

Note

The `spring-milestones` and `spring-snapshots` repositories are available by default. Please note that these are just references and won't impact the project unless you choose a dependency that explicitly refer to a bom and/or repo by id. Check the example below for more details.

4.2 Dependencies section

The `dependencies` section allows you define a list of groups, each group having one more dependency. A group gather dependencies that share a common characteristics (i.e. all web-related dependencies for instance).

A dependency has the following basic characteristics:

- A mandatory identifier. If no further information is provided, a Spring Boot starter with that id is assumed.
- A name and description used in the generated meta-data and the web ui.
- A `groupId` and `artifactId` to define the coordinates of the dependency.
- A `version` if Spring Boot does not already provide a dependency management for that dependency.
- A `scope` (can be `compile`, `runtime`, `provided` or `test`).
- The reference to a `bom` or a `repository` that must be added to the project once that dependency is added.
- A `versionRange` used to determine the Spring Boot versions that are compatible with the dependency.
- Links to resources such as a guide or a reference doc section.

Tip

Check [the code](#) for a full list of the available configuration options.

Here is the most basic dependency entry you could have

```
initializr:
  dependencies:
    - name: Core
      content:
        - id: security
          name: Security
          description: Secure your application via spring-security
```

Tip

The security dependency is held within a group called "Core".

This adds an option name `Security` with a tooltip showing the description above. If a project is generated with that dependency, the `org.springframework.boot:spring-boot-starter-security` dependency will be added to the project.

Let's now add a custom dependency that is not managed by Spring Boot and that only work from Spring Boot 1.2.0.RELEASE and onwards but should not be used in the 1.3 lines and further for some reason.

```
initializr:
  dependencies:
    - name: Core
      content:
        - id: my-lib-id
          name: My lib
          description: Secure your application via spring-security
          groupId: com.example.foo
          artifactId: foo-core
          bom: foo-bom
          repository: foo-repo
          versionRange: "[1.2.0.RELEASE,1.3.0.M1)"
```

If one selects this entry, the `com.example.foo:foo-core` dependency will be added and the Bill of Materials and repository for `foo` will be added automatically to the project as well (see the "Env section" above for a reference to those identifiers). Because the bom provides a dependency management for `foo-core` there is no need to hard code the version in the configuration.

The `versionRange` syntax follows some simple rules: a square bracket "[" or "]" denotes an inclusive end of the range and a round bracket "(" or ")" denotes an exclusive end of the range. A range can also be unbounded by defining a single version. In the example above, the dependency will be available as from `1.2.0.RELEASE` up to, not included, `1.3.0.M1` (which is the first milestone of the 1.3 line).

Dependency group

A dependency group gather a set of dependencies as well as some common settings: `bom`, `repository` and `versionRange`. If one of them is set, it is applied for all dependencies within that group. It is still possible to override a particular value at the dependency level.

4.3 Other sections

The other section defines the default and the list of available options in the web UI. This also drives how the meta-data for your instance are generated and tooling support is meant to react to that.

For instance, if you want your `groupId` to default to `org.acme` and the `javaVersions` to only be `1.7` and `1.8` you would write the following config:

```
initializr:
  groupId:
    value: org.acme
  javaVersions:
    - id: 1.8
      default: true
    - id: 1.7
      default: false
```