Project Title: TSE Initializr

Every good Spring Boot project usually starts at https://start.spring.io/

While this probably holds true for thousands of software projects every year, the app behind the scenes, the so-called Spring Initialzr, is Open Source as well and can even be customized to your own needs.

Spring Initializr enables us to create new Spring Boot development projects. The site offers the possibility to configure certain aspects of the project, such as its Maven coordinates or base package, and provides the developer with an extensive list of dependencies to choose from. Many of them are Spring Boot Starters in fact.

A Brief Introduction of Spring Boot Starter

Spring Boot takes an opinionated view of the Spring platform to enable developers to build stand-alone, production-ready applications with as little effort as possible.

A key success factor to achieve this is convention over configuration. Instead of having to configure Spring or any 3rd-party components, such as databases and message brokers, manually, Spring provides out-of-the-box auto configurations following common configuration conventions.

Separate code, despite some @Enable annotations, isn't required, unless manual overriding of the configuration is necessary.

These auto configurations are usually shipped as Spring Boot Starters, which focus on configuring a certain aspect of an application, typically by means of bean configurations for both Spring internal and 3rd-party components.

The Spring Boot Web Starter is a very popular one and configures the Spring MVC framework as well as an embedded Tomcat, Jetty, or Undertow server, so the application is able to start its own web server.

Source: https://start.spring.io

Upon project generation an archive is created, which can then be downloaded to start with the development. The archive itself does not include any binary dependencies itself, but simply reflects the previously chosen project configuration, either as Maven or Gradle build file.

Spring Initializr components From Development Standpoint:

The Spring Initalizr website is a Spring Boot application itself. The following diagram gives an overview of the different components that make up the Spring Initializr.

1. The core Spring Initializr
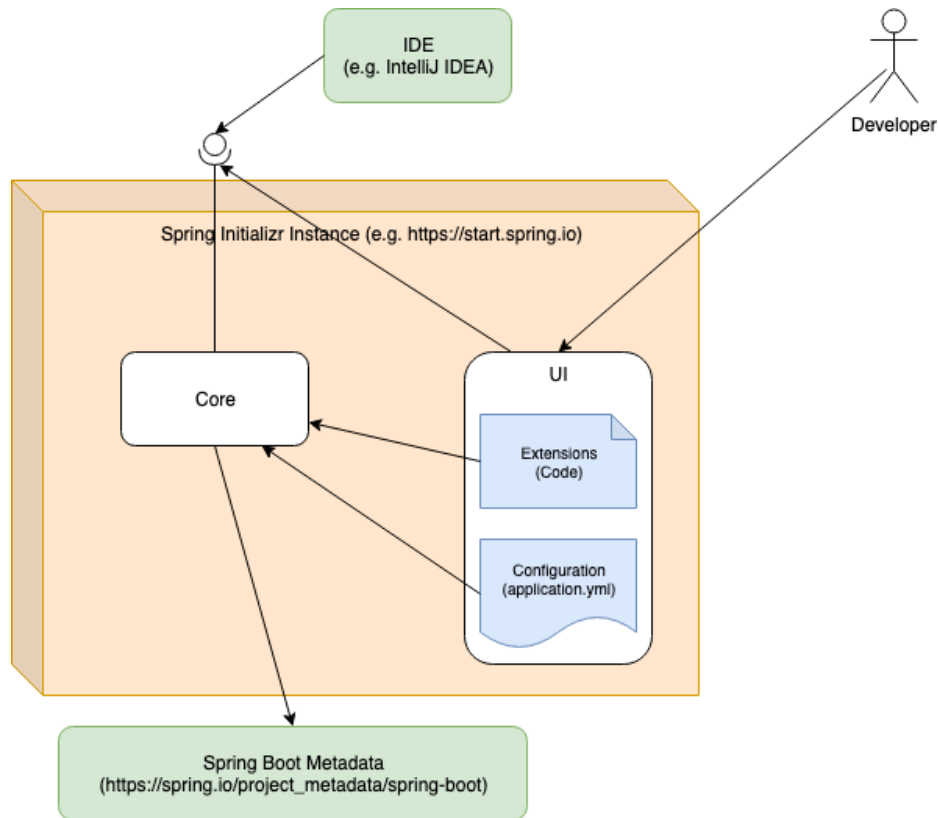
   https://github.com/spring-io/initializr

   provides the generation logic as well as a REST API. The latter one is used for integration into the UI or popular IDEs such as IntelliJ IDEA, which supports project generation directly from a Spring Initializr instance.

2. The UI

   https://github.com/spring-io/start.spring.io

   provides the well-known web interface and comprehensive configuration for the project metadata and dependencies included therein. The UI heavily relies on the core Spring Initializr and its REST APIs. As such, the configuration that is part of the UI project is not for the UI itself, but rather configures the project metadata provided via the REST API. All this is achieved using standard Spring Boot properties e.g., application.yml, and customization code based on the extension hooks provided by the core, as we will see later in this post.

3. In addition, Spring.io provides a Spring Boot metadata endpoint, which is consumed by the core Spring Initializr as an external resource, to ensure the Spring Boot versions provided during project generation are always up to date.

Problem Description

While https://start.spring.io/ is an excellent start for your next Spring Boot project, sometimes it may be necessary to customize it to your own needs e.g., for the following reasons:

1. You may be required to provide a self-hosted instance of the Spring Initializr within your company e.g., due to network restrictions when downloading archives from the web.

2. You may want to tweak or brand the UI or even build your own UI from scratch.

3. You may want to provide your own project configurations and/or dependencies e.g., company internal Spring Boot starters that are not publicly available on the web.

Due to the modular architecture of the Spring Initializr and the fact that it is a regular Spring Boot application, there are various approaches for extending it:

1. You may choose to go with the Spring Initializr core and build your own UI on top of it, if needed. This approach involves providing suitable dependency configurations and extension code as well, which is well suited for cases, where you want to have full control of them.

2. You may choose to go with the Spring Initializr UI selectively adapting its configuration, extension code, and the UI (if needed). This approach gives you access to a huge list of predefined dependencies and customizations e.g., Spring Cloud, Messaging, and Data (see link application.yml below). All of them are maintained by the project contributors. You may extend the configuration and extension code to provide your own dependencies on top, if needed, or you may simple adapt the UI to your own needs.

application.yml link

https://github.com/maverick1601/start.spring.io/blob/blog_post/start-site/src/main/resources/application.yml

Second approach, as it inherits the existing configuration from Spring.IO.

Building your own TSE Initializr

When choosing to build your own TSE Initializr based on the Spring Initializr UI, the biggest challenge is to avoid conflicts when updating the original project from GitHub. Remember, that the Initializr configuration is frequently updated, because it has to reflect newly available versions of the dependencies. Thus, direct modification of the project's application.yml may result in repeating conflicts, when merging updates from the master branch so unless you decide to publicly submit your custom dependencies, a separate configuration approach is required.

Before getting our hands dirty, we first need to clone and build the two Spring Initializr components in the following order:

1. https://github.com/spring-io/initializr
2. https://github.com/spring-io/start.spring.io

Building the core is necessary, because it's not available from Maven Central. For the sake of simplicity, the customizations shown in this blog post will be directly applied to the UI project, more explicitly to the start-site Maven module contained therein. You may as well choose to create a new Maven module or a separate project.

Custom Initializr Configuration

To add custom dependencies to the configuration without unnecessary modification of the existing files, we cannot simply create a second application.yml. While it is possible, in general, to load configuration properties from multiple locations, they won't be merged, when it comes to list or map structures, for good reasons. The InitializrProperties(available on GitHub), which provide the configuration for the core, heavily rely on such collections, e.g. for defining the list of available dependencies.

Either one of the different application.yml files would have to define the complete set of dependencies then.

Fortunately, Spring Initializr's auto configuration allows us to define a custom InitializrMetadataProvider(available on GitHub), which in turn enables us to merge multiple InitializrProperties(available on GitHub). Hence, we can create a CustomInitializrProperties(available on GitHub) class, which uses a prefix for separating our own properties from the provided defaults. The following Spring configuration creates the metadata provider after merging these two property sets.

CustomInitializrConfiguration enabling merged Spring Initializr property configuration

The configuration needs to be hooked into StartApplication, since the application class does not use component scanning. Now, we can easily define our own configuration elements, prefixed with custom:, e.g. by placing them in a separate application.yml in the config package, as follows:

application.yml based Spring Initializr configuration for custom dependencies

With this additional custom dependency in place, once we start the application, we can now choose from the merged set of dependencies. The order of loading and merging the two InitializrProperties was chosen, such that our custom dependency group is placed in front of the standard dependencies, as shown in the following example:

Spring Initializr with custom dependencies added

While this approach enables us to provide our own custom dependencies, still being able to update from the master branch on a regular basis, it does not offer any way of suppressing (or overwriting) standard dependencies. In the end, this is simply a cumulative merge of the two sets of configuration.

Custom Initializr Extensions

Adding custom dependencies via configuration isn't always sufficient for project generation. Sometimes additional extensions need to take care of customizing the project during its generation phase. Spring Initializr core provides different extension hooks for such cases:

1. BuildCustomizer implementations can be used to customize the Maven/Gradle build, e.g. adding more dependencies or build plugins.
2. ProjectContributor implementations can be used to contribute additional contents, i.e. files and directories, to the project structure.

3. MainSourceCodeCustomizer, MainCompilationUnitCustomizer, MainApplicationTypeCustomizer, TestSourceCodeCustomizer, TestApplicationTypeCustomizer implementations support the generation/modification of source code within the project, independently of the selected programming language.
4. GitIgnoreCustomizer enables the customization of the .gitignore file.
5. HelpDocumentCustomizer can be used to extend the HELP.md file.
6. ProjectDescriptionCustomizer implementations are typically used to adapt the project description, for instance automatically resolving invalid combinations of framework versions and language levels.

Suppose we want to add a Maven plugin to our generated project, a BuildCustomizer<MavenBuild> implementation can be used to add the plugin including its configuration. However, the plugin shall not be added for all projects. Hence, we need to use a so-called *pseudo dependency* (see: https://github.com/spring-io/initializr/issues/915), that is a dependency, which we remove during the project generation right after we've added the plugin configuration. Accordingly, we use the well known property based configuration to define such custom Maven plugin dependency, as shown in the following example:

application.yml for custom Maven plugin dependency

Next we need to implement two BuildCustomizers: one for adding the Maven plugin, another one for removing the plugin dependency afterwards, so it won't get added to the actual list of project dependencies. These customizers need to be defined as Spring beans within a ProjectGenerationConfiguration class, as shown in the following example:

CustomMavenPluginConfiguration configuring Maven plugin in pom.xml

While this class looks like a regular Spring configuration class, it won't be picked up as part of the Spring Initializr application context, but will be instantiated on-demand for each project generation request. That's why the class can be annotated using special dependency selection conditions (for instance ConditionalOnRequestedDependency).

Moreover, this is also the reason, why the StartApplication does not use component scanning, as it would accidentally pick up such project generation configurations.

For the Spring Initializr to know, which configurations need to be taken into considerations for project generations, they need to be registered via spring.factories, e.g. as follows:

CustomMavenPluginConfiguration registration

Finally, the plugin pseudo dependency can be used to create Maven projects, which then include the following plugin configuration as part of their pom.xml:

User Stories:

1. The application should allow us to add dependencies

2. The application should allow to choose options between war and jar

3. The application should allow us to choose the appropriate jdk version

4. The application us to mention the domain, artifact, name of project and the description of a project

5. The application should allow us to choose maven or Gradle

6. Rest response should be successful to get the accurate dependencies from Maven or Gradle

7. zip file should be created properly and saved on system

8. Lastly, we should be able to load the project in Intellij or Eclipse or any IDE.