Spring-Boot

- Architecture, Social Content, Pull Request/Issues

Introduction

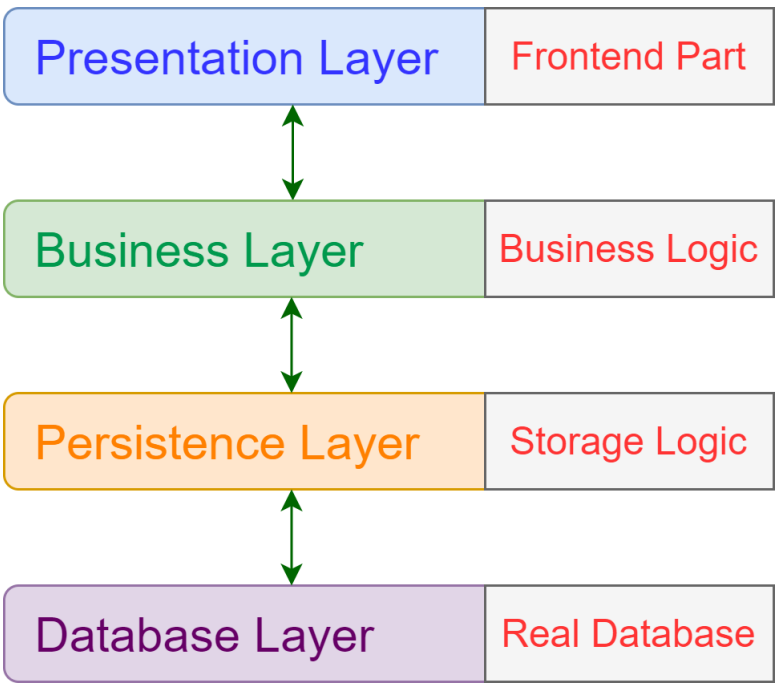
Spring is widely used to develop scalable applications, and for web application, among all the modules provided by Spring, Spring MVC is widely used for this specific application. However, like we discussed before, spring development is extremely time-consuming and overwhelming for beginners. And Spring-Boot is the cure for this problem. Spring-boot is built on top of spring and integrate many packages and make development much easier than Spring.

In this document, we discuss the architecture, social content and some interesting pull request and issues about spring-boot.

Architecture

Because Spring-boot is built on top of the Spring, Spring-boot has nearly all spring features and its architecture is nearly same with spring MVC. They both follow the ***Layered Architecture***. Spring-boot contains 4 layers.

- Presentation Layer
- Business Layer
- Persistence Layer
- Database Layer



1. Presentation Layer

Presentation Layer is in charge of handling HTTP requests. After receiving responses, it parses JSON to objects or models, and transfer it to the Business Layer. In short, it deals with frontend parts. For example, `Dispatcher`, `HandlerMapper`, `MustacheViewResolver` are classes in the presentation layer, and they implement or extend classes defined in Spring.

2. Business Layer

Business handles all the business logic in our application. It consists of service classes and user services provided by persistence layer. And if application needs authorization or validation, it's performed in this layer as well. This layer is implemented by users. For example, if user needs to monitor system runtime status, user could use endpoints provided by spring-boot, and control the application with designed logic.

3. Persistence Layer

This layer works as a filter before the database layer. Most of times, response would contain many unrelated information, or information that don't need to save in the database. This is also implemented by user. In short, persistence layer contains all the storage logic and translates business objects from and to the database.

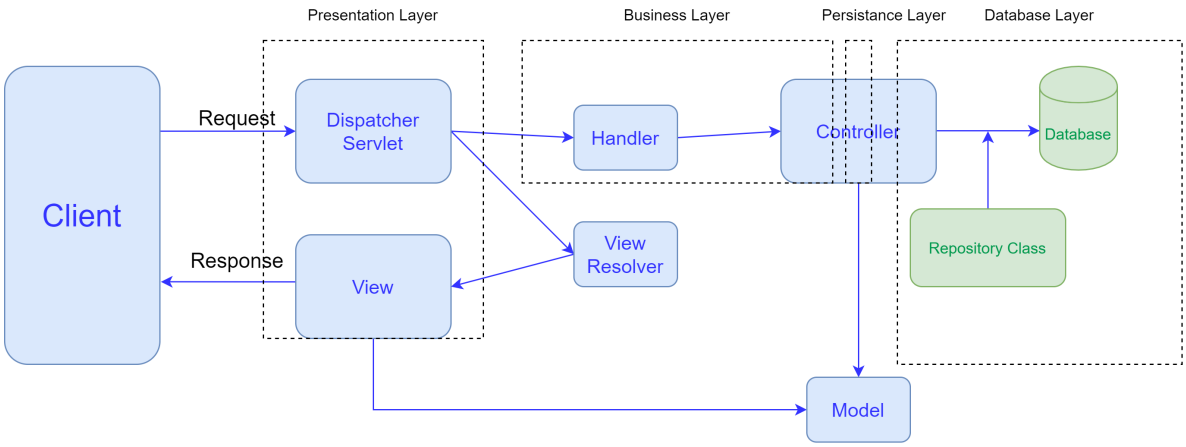
4. Database Layer

In the database layer, Create, retrieve, update and delete operations are performed. In spring-boot, it provides `jdbc` and `cassandra` interface to work as database.

Flow Architecture

To clarify at first, client refers to the actual server that publish HTTP request, and server refers to the actual server that make the response.

Behind the layer architecture, due to the reason that spring-boot is actually an encapsulation of spring, spring-boot follows the ***MVC architecture*** as well.



At first, client makes an HTTP request(PUT/GET), our server, `DispatcherServlet` receives that request, and it dispatches the task of selecting `controller` to `HandlerMapping`. `HandlerMapping` selects the `controller` and return the `handler` and `controller` to `DispatcherServlet` .

```
public class Dispatcher {
...
private final List<HandlerMapper> mappers;
public boolean handle(ServerHttpRequest request, ServerHttpResponse
response) throws IOException {
```

```
        for (HandlerMapper mapper : this.mappers) {
            Handler handler = mapper.getHandler(request);
            if (handler != null) {
                handle(handler, request, response);
                return true;
            }
        }
        return false;
    }
...
}
```

> ***Code Explanation***
>
> This part of code shows the details of `Dispatcher`. `mapper` contains the right `handler` for specific request. If we do have a handler for the request, we call handle function to deal with these request.

After the `controller` and `handler` are assigned, `DispatcherServlet` will send the task of executing business logic to `controller` and `HandlerAdapter`. `HandlerAdapter` will call the business logic and `controller` will execute it. After it's executed, `controller` will set the processing results to database and also send them to `Model`. `HandlerAdapter` is defined in Spring, not Spring-boot, so we didn't put it in the diagram. To dive into this process, we look it up in the Spring source code. The actual implementation is as follows.

```
public class HttpRequestHandlerAdapter implements HandlerAdapter {
    ...
    public ModelAndView handle(HttpServletRequest request,
HttpServletResponse response, Object handler)
            throws Exception {

        ((HttpRequestHandler) handler).handleRequest(request, response);
        return null;
    }
```

```
public abstract class AbstractUrlViewController extends AbstractController
{
    ...
    protected ModelAndView handleRequestInternal(HttpServletRequest
request, HttpServletResponse response) {
        String viewName = getViewNameForRequest(request);
        if (logger.isTraceEnabled()) {
            logger.trace("Returning view name '" + viewName + "'");
        }
        return new ModelAndView(viewName,
RequestContextUtils.getInputFlashMap(request));
    }
    ...
}
```

> ***Code Explanation***

> This code is implemented in spring framework. Firstly, `Adapter` will handle the request and return `model` and view name. These values would be used in `DispatcherServlet` . The request is actually implemented in `Controller`. After business is executed, `model` and view name are returned by `Controller` and then returned by `Adapter`.

Now, `ViewResolver` will resolve the view name returned by `HandlerAdapter` and `DispatcherServlet` will dispatch the rendering process to `View`. `View` is in charge of rendering `Model` and return the response, JSP for instance.

Social Content

Spring Boot is an open source Java-based framework used to create a micro Service. It is developed by Pivotal Team and is used to build stand-alone and production ready spring applications. On GitHub, it has been stared 45.6k times and forked 28.8k times.

State of the Project

Spring-boot came to world in 2012 and has been widely used in developing. The birth of spring boot sets the only road map of spring boot. A client of Pivotal complained about the difficulty of developing applications with Spring. And that's how spring boot came to world.

The roadmap for Spring Boot is to build an easily use and light framework. Spring-Boot describes their goals as follows.

- Provide a radically faster and widely accessible getting started experience for all Spring development
- Be opinionated out of the box, but get out of the way quickly as requirements start to diverge from the defaults
- Provide a range of non-functional features that are common to large classes of projects (e.g. embedded servers, security, metrics, health checks, externalized configuration)
- Absolutely no code generation and no requirement for XML configuration

Here are other information about state of Spring Boot.

- Last released date: Feb.13 2020
- Recent commits: 25373 till now
- Issues: open - 419, closed - 15800
- Active Developer: 656 direct contributors

Project Standards

Code Standard

Spring Boot nearly passed the Sun style check. However, there are 28 lines that are not followed by Sun Standard. What's more, it fails 128 lines on Google style check.

Testing Standard

Spring Boot uses JUNIT and performs unit test on every class.

Document Standard

The official document is saved here. And the tutorial on GitHub is set to that link on the official website as well.

Commit Standard

Spring Boot doesn't have a specific commit standard, and I think we should follow the regular GitHub work flow, which is fork, change, and submit a pull request.

Contribution Process

Because Spring Boot is a open source project, all issues and tasks are done on issues section on GitHub. To contribute the project, we should follow these process. We fork it, make changes, and submit a pull request. Also, Spring Boot has a channel on Gitter, we could answer questions on it as well. Spring Boot also has a support team on stack overflow, where we could contribute as well.

To identify a bug, developers ask us to provide the full project to reproduce the bug. But we should first search on issue tracker, then if the bug is indeed never discovered, we could open a new issue.

Contribution Tools

GitHub, Gitter, StackOverflow, and we study TravisCI on another class, it is a great tool for building and testing projects on different environments and every branch.

Pull Requests

Five Interesting Pull Requests

1. Pull request #20173:

This pull request was created just to document that the Spring Boot actuator over HTTP requires Jackson. This is an interesting pull request because it seemed to accurately resolve the reported issue, but it was declined because the issue was already assigned to another contributor. It's interesting that the developers would only allow one person to resolve the issue rather than fixing it as soon as possible.

2. Pull request #19959:

This pull request was created to add functionality for Spring Boot to support shared jar launcher design implementations. This is an interesting pull request because the developers recognized that this could be a helpful feature. However, after further discussion, they decided to decline the pull request since there was not enough demand for the feature to justify the cost of maintaining the feature.

3. Pull request #19677:

This pull request aims to add functionality for the setDefaultProperties() method in the app class to support changes to immutable maps. This is an interesting request because if implemented, it could potentially cause more issues for users that need an immutable map to remain immutable. The developers realized this and decided to maintain the original functionality. They also provided additional options for the user to solve the specific issue they were experiencing, such as passing a mutable map instead of an immutable one.

4. Pull request #18707:

This pull request was created in an attempt to update Spring Boot to be dependent on Byte buddy 1.10.2 rather than the previous version. This is an interesting pull request because it goes against one of the rules clearly discussed in the pull request template for this project. Pull requests are not accepted for upgrades of Spring Boot's dependencies.

5. Pull request #18678:

This pull request was created in an attempt to add new auto-configure properties for JMs. The contributor was under the impression that they were making auto-configuration easier. However, the developers recognized that these changes could actually hinder the auto-configuration functionality, which is an essential feature of Spring Boot. Therefore, this pull request was ultimately declined.

Five Interesting Issues

In this section, we will discuss some interesting issues.

1. Enhancements for new LAYERED_JAR layout in spring-boot-maven-plugin

This issue was opened by a senior engineer. She wants to give some suggestions for the new maven plugin.

She suggests that

1. Layers should be configurable. In her company, there is a update cycle dependencies and organization dependencies, so the latter should be positioned in an upper layer.
2. She thinks that it's weird to package a jar in LAYERED_JAR format. Because it will be used for building images, not executing.
3. Spring Boot: building image could be launched standalone, but layer jar depends on the package phase, and it's executed after maven-jar-plugin. As a result, the original jar will be built on repo.

It's always interesting to know the ideas in a senior SDE. How they think and how they resolve the problem, this concern arises in a particular industry circumstances that I have no access to. Developers of Spring Boot address this concern quickly. They suggest that

1. In the future release, customizable layer could be added.
2. Layered jars and jib are largely equivalent and achieve a similar end goal. Any major pros and cons are likely to be down to personal preference. Without knowing what you like and dislike about jib, it's difficult to answer.

   The buildpack support is a different take on things. It allows you to build an image in the same way a Platform would. One advantage of this is that changes can be made in a centralised place (the builder) and all applications using the builder will then pick up that change.

And asker gets a better understanding about the design of layer jar and the issue is closed.

2. Outdated library prevents server startup

This issue was opened on Feb. 21 2020. Their Spring Boot application allows to be extended by users who add their own jars (and dependencies) to the classpath. If an outdated library is added (e.g. GSON 2.3.1), server startup fails with a `BeanCreationException`.

It's interesting that we use GSON in lots of places and another engineer has the same problem. And how developer analyzes the problem is really helpful for us. The developer addressed that `BeanCreationException` has a more broad meaning and they couldn't add some codes to avoid it in auto-configuration. Then asker provided the project for reproducing the issue, and developer marked this `team: attention` . Then developer gave a solution to this problem, which is to back off if an older version is on the classpath (via `@ConditionalOnClass`) or where we catch `NoSuchMethodError` to tolerate older versions.

However, this could only solve the problem with GSON, not other conditions. And answerer suggests that auto-configuration is designed to make sure every package is working, and this problem could only be solved case by case for now.

3. Actuator doesn't use the CORS Configuration with default security config

This issue was opened on Feb.9 2018. She suggests that If Spring Security can add this as a default, we don't need to do anything. If Spring Security doesn't make this a default, we need to see how this can be done. The original solution won't work for Jersey because there is no `CorsFilter` or `CorsConfigurationSource`.

Firstly, the developer suggested that adding this feature would tangle actuator and security, but the asker made this explanation.

> If Spring Security added it as a default, Spring Boot wouldn't need to add anything extra and it would just rely on Spring Security's defaults. It wouldn't tangle Actuator and Security because there would be no actuator specific configuration in Spring Boot's security auto-config.

Developers adopt this idea and add to milestone for further discussion. This issue is still open, maybe there's some controversy about this problem.

4. Consider adding a `provider.base-url` property under OAuth client properties

This issue was opened on Dec.28 2017. As we know, to send a HTTP request, we need to build our url with `authorizationUri`, `tokenUri`, `userInfoUri` and `jwkSetUri` . But the base url is normally same, so by adding a provider.base-url, it could reduce the verbosity.

Snicoll(developer) wrote a psedo code and it ends up with more verbosity.

```
keycloak-client.base-uri=${keycloak-client.server-url}/realms/${keycloak-client.realm}/protocol/openid-connect
spring.security.oauth2.client.provider.keycloak.authorizationUri=${keycloak-client.base-uri}/auth
spring.security.oauth2.client.provider.keycloak.jwkSetUri=${keycloak-client.base-uri}/certs
spring.security.oauth2.client.provider.keycloak.tokenUri=${keycloak-client.base-uri}/token
spring.security.oauth2.client.provider.keycloak.userInfoUri=${keycloak-client.base-uri}/userinfo
```

This is interesting because somtimes things are not working as we expected. Asker explained that she intended to replace `spring.security.oauth2.client.provider.keycloak.userInfoUri` with `base-uri/userinfo`. However, that led to modification for userInfoUrl, but Snicoll thought that shouldn't be editable but only sharable. Asker didn't give a response since then.

5. configprops endpoint cannot display a ConfigurationProperties bean that is a List

This issue was opened on Feb.26 2020. This is a pretty recent issue and solved very quickly.

Asker noticed that in yaml configuration, application has this codes:

```
datasource:
  transaction:
    managers:
      - transactionManagerA
      - transactionManagerB
      - transactionManagerC
      - transactionManagerD
```

And it was injected into a class like this:

```
@Configuration
public class MultipleDatasourceTransactionManager {
    private static final Logger LOGGER =
LoggerFactory.getLogger(MultipleDatasourceTransactionManager.class);
    public static final String BEAN_ID_LIST_OF_TRANSACTION_MANAGERS =
"listOfTransactionManagers";
    public static final String BEAN_ID_TRANSACTION_MANAGER =
"transactionManager";

    @Autowired
    private ApplicationContext context;

    @Bean (name = BEAN_ID_LIST_OF_TRANSACTION_MANAGERS)
    @ConfigurationProperties (prefix = "datasource.transaction.managers")
    public List<String> listOfTransactionManagers() {
        return new ArrayList<>();
    }
   ...
```

But an error occurs: "error": "Cannot serialize 'datasource.transaction.managers'".

Developer agreed with him and believed that list binding is not supported by binding. However, if using POJO for propertied, it could work for binding "by accident". Instead of passing a list, by passing a POJO, the problem would be solved.

```
System.out.println(new SpringApplicationBuilder(Gh20319Application.class)

.properties("datasource.transaction.managers[0]=transactionManagerA",

 "datasource.transaction.managers[1]=transactionManagerB",

 "datasource.transaction.managers[2]=transactionManagerC",

 "datasource.transaction.managers[3]=transactionManagerD")

.run().getBean(DatasourceTransactionProperties.class).getManagers());
```

Reference

1. https://www.javatpoint.com/spring-boot-architecture
2. https://spring.io/projects/spring-boot
3. https://spring.io/support
4. https://github.com/spring-projects/spring-boot