# 1.[How to configure port for a Spring Boot application](https://stackoverflow.com/questions/21083170/how-to-configure-port-for-a-spring-boot-application)

server.port = 8090

For random port use

server.port=0

# 2.[how to log sql statements in spring-boot](https://stackoverflow.com/questions/30118683/how-to-log-sql-statements-in-spring-boot)

spring.jpa.properties.hibernate.show\_sql=true

spring.jpa.properties.hibernate.use\_sql\_comments=true

spring.jpa.properties.hibernate.format\_sql=true

To log values:

spring.jpa.properties.hibernate.type=trace

# 3.[How to add a filter class in Spring Boot?](https://stackoverflow.com/questions/19825946/how-to-add-a-filter-class-in-spring-boot)

There are three ways to add your filter,

1. Annotate your filter with one of the Spring stereotypes such as @Component
2. Register a @Bean with Filter type in Spring @Configuration
3. Register a @Bean with FilterRegistrationBean type in Spring @Configuration

Either #1 or #2 will do if you want your filter applies to all requests without customization, use #3 otherwise. You don't need to specify component scan for #1 to work as long as you place your filter class in the same or sub-package of your SpringApplication class. For #3, use along with #2 is only necessary when you want Spring to manage your filter class such as have it auto wired dependencies. It works just fine for me to new my filter which doesn't need any dependency autowiring/injection.

Although combining #2 and #3 works fine, I was surprised it doesn't end up with two filters applying twice. My guess is that Spring combines the two beans as one when it calls the same method to create both of them. In case you want to use #3 alone with authowiring, you can AutowireCapableBeanFactory. The following is an example,

private @Autowired AutowireCapableBeanFactory beanFactory;

@Bean

public FilterRegistrationBean myFilter() {

FilterRegistrationBean registration = new FilterRegistrationBean();

Filter myFilter = new MyFilter();

beanFactory.autowireBean(myFilter);

registration.setFilter(myFilter);

registration.addUrlPatterns("/myfilterpath/\*");

return registration;

}

# 4. [How to access a value defined in the application.properties file in Spring Boot](https://stackoverflow.com/questions/30528255/how-to-access-a-value-defined-in-the-application-properties-file-in-spring-boot)

You can use the @Value annotation and access the property in whichever Spring bean you're using

@Value("${userBucket.path}")

private String userBucketPath;

Another way is injecting Environment to your bean.

@Autowired

private Environment env;

....

public void method() {

.....

String path = env.getProperty("userBucket.path");

.....

}

# 5.[What is the purpose of mvnw and mvnw.cmd files?](https://stackoverflow.com/questions/38723833/what-is-the-purpose-of-mvnw-and-mvnw-cmd-files)

These files are from [Maven wrapper](https://github.com/takari/maven-wrapper). It works similarly to the [Gradle wrapper](https://docs.gradle.org/current/userguide/gradle_wrapper.html).

This allows you to run the Maven project without having Maven installed and present on the path. It downloads the correct Maven version if it's not found (as far as I know by default in your user home directory).

The mvnw file is for Linux (bash) and the mvnw.cmd is for Windows environment.

To create or update all necessary Maven Wrapper files execute the following command:

mvn -N io.takari:maven:wrapper

To use a diffrent version of maven you can specify the version as follows:

mvn -N io.takari:maven:wrapper -Dmaven=3.3.3

Both commands require maven on PATH, if you already have mvnw in your project you can use ./mvnw instead of mvn in the commands.

# 6. [Add context path to Spring Boot application](https://stackoverflow.com/questions/20405474/add-context-path-to-spring-boot-application)

If you are using Spring Boot, then you don't have to configure the server properties via Vean initializing.

Instead, if one functionality is available for basic configuration, then it can be set in a "properties" file called application, which should reside under src\main\resources in your application structure. The "properties" file is available in two formats

1. .yml
2. .properties

The way you specify or set the configurations differs from one format to the other.

In your specific case, if you decide to use the extension .properties, then you would have a file called application.properties under src\main\resources with the following configuration settings

server.port = 8080

server.contextPath = /context-path

OTOH, if you decide to use the .yml extension (i.e. application.yml), you would need to set the configurations using the following format (i.e. YAML):

server:

port: 8080

contextPath: /context-path

# 7. [Spring Boot REST service exception handling](https://stackoverflow.com/questions/28902374/spring-boot-rest-service-exception-handling)

My project has the same requirements as yours: I want my REST API to return a HTTP 404 Not Found with an accompanying JSON payload in the HTTP response to the API client when it tries to send a request to an URL which does not exist. In my case, the JSON payload looks like this (which clearly differs from the Spring Boot default, btw.):

{

"code": 1000,

"message": "No handler found for your request.",

"timestamp": "2017-11-20T02:40:57.628Z"

}

I finally made it work. Here are the main tasks you need to do in brief:

* Make sure that the NoHandlerFoundException is thrown if API clients call URLS for which no handler method exists (see Step 1 below).
* Create a custom error class (in my case ApiError) which contains all the data that should be returned to the API client (see step 2).
* Create an exception handler which reacts on the NoHandlerFoundException and returns a proper error message to the API client (see step 3).
* Write a test for it and make sure, it works (see step 4).

Ok, now on to the details:

**Step 1: Configure application.properties**

I had to add the following two configuration settings to the project's application.properties file:

spring.mvc.throw-exception-if-no-handler-found=true

spring.resources.add-mappings=false

This makes sure, the NoHandlerFoundException is thrown in cases where a client tries to access an URL for which no controller method exists which would be able to handle the request.

**Step 2: Create a Class for API Errors**

I made a class similar to the one suggested in [this article](http://www.baeldung.com/global-error-handler-in-a-spring-rest-api) on Eugen Paraschiv's blog. This class represents an API error. This information is sent to the client in the HTTP response body in case of an error.

public class ApiError {

private int code;

private String message;

private Instant timestamp;

public ApiError(int code, String message) {

this.code = code;

this.message = message;

this.timestamp = Instant.now();

}

public ApiError(int code, String message, Instant timestamp) {

this.code = code;

this.message = message;

this.timestamp = timestamp;

}

// Getters and setters here...

}

**Step 3: Create / Configure a Global Exception Handler**

I use the following class to handle exceptions (for simplicity, I have removed import statements, logging code and some other, non-relevant pieces of code):

@RestControllerAdvice

public class GlobalExceptionHandler {

@ExceptionHandler(NoHandlerFoundException.class)

@ResponseStatus(HttpStatus.NOT\_FOUND)

public ApiError noHandlerFoundException(

NoHandlerFoundException ex) {

int code = 1000;

String message = "No handler found for your request.";

return new ApiError(code, message);

}

// More exception handlers here ...

}

**Step 4: Write a test**

I want to make sure, the API always returns the correct error messages to the calling client, even in the case of failure. Thus, I wrote a test like this:

@RunWith(SpringRunner.class)

@SpringBootTest(webEnvironment = SprintBootTest.WebEnvironment.RANDOM\_PORT)

@AutoConfigureMockMvc

@ActiveProfiles("dev")

public class GlobalExceptionHandlerIntegrationTest {

public static final String ISO8601\_DATE\_REGEX =

"^\\d{4}-\\d{2}-\\d{2}T\\d{2}:\\d{2}:\\d{2}\\.\\d{3}Z$";

@Autowired

private MockMvc mockMvc;

@Test

@WithMockUser(roles = "DEVICE\_SCAN\_HOSTS")

public void invalidUrl\_returnsHttp404() throws Exception {

RequestBuilder requestBuilder = getGetRequestBuilder("/does-not-exist");

mockMvc.perform(requestBuilder)

.andExpect(status().isNotFound())

.andExpect(jsonPath("$.code", is(1000)))

.andExpect(jsonPath("$.message", is("No handler found for your request.")))

.andExpect(jsonPath("$.timestamp", RegexMatcher.matchesRegex(ISO8601\_DATE\_REGEX)));

}

private RequestBuilder getGetRequestBuilder(String url) {

return MockMvcRequestBuilders

.get(url)

.accept(MediaType.APPLICATION\_JSON);

}

The @ActiveProfiles("dev") annotation can be left away. I use it only as I work with different profiles. The RegexMatcher is a custom [Hamcrest matcher](http://hamcrest.org/JavaHamcrest/) I use to better handle timestamp fields. Here's the code (I found it [here](http://www.vogella.com/tutorials/Hamcrest/article.html)):

public class RegexMatcher extends TypeSafeMatcher<String> {

private final String regex;

public RegexMatcher(final String regex) {

this.regex = regex;

}

@Override

public void describeTo(final Description description) {

description.appendText("matches regular expression=`" + regex + "`");

}

@Override

public boolean matchesSafely(final String string) {

return string.matches(regex);

}

// Matcher method you can call on this matcher class

public static RegexMatcher matchesRegex(final String string) {

return new RegexMatcher(regex);

}

}

**Some further notes from my side:**

* In many other posts on StackOverflow, people suggested setting the @EnableWebMvcannotation. This was not necessary in my case.
* This approach works well with MockMvc (see test above).

# 8. [Spring Boot Configure and Use Two DataSources](https://stackoverflow.com/questions/30337582/spring-boot-configure-and-use-two-datasources)

Here you go

#first db

spring.datasource.url = [url]

spring.datasource.username = [username]

spring.datasource.password = [password]

spring.datasource.driverClassName = oracle.jdbc.OracleDriver

#second db ...

spring.secondDatasource.url = [url]

spring.secondDatasource.username = [username]

spring.secondDatasource.password = [password]

spring.secondDatasource.driverClassName = oracle.jdbc.OracleDriver

@Bean

@Primary

@ConfigurationProperties(prefix="spring.datasource")

public DataSource primaryDataSource() {

return DataSourceBuilder.create().build();

}

@Bean

@ConfigurationProperties(prefix="spring.secondDatasource")

public DataSource secondaryDataSource() {

return DataSourceBuilder.create().build();

}