infogain

SPRING BOOT

Part2

Objective

- Spring Boot Change Context Path and Server Port
- Spring Boot and JdbcTemplate
- Spring Boot and Internalization
- Deploy WAR file to Tomcat

Spring Boot Change Context Path and Server Port

- **There are several ways to change default context path.**
  - Using Property File (.properties)
  - Using SERVER_CONTEXT_PATH with SpringApplication Programmatically
  - Using EmbeddedServletContainerCustomizer

- **Using Property File (application.properties)**
  - server.port=8088
  - server.contextPath = /spring-boot

SpringApplication Class Programmatically

- **SpringApplication** has a method as **setDefaultProperties()** that is used to change spring boot default properties.
- Create a Map object and use below key value pair
- SERVER_CONTEXT_PATH key with value of desired context path name using prefix ("/").
- SERVER_PORT key with desired port value

Application.java

- SpringApplication application = new SpringApplication(Application.class);
- Map<String, Object> map = new HashMap<>();
- map.put("SERVER_CONTEXT_PATH", "/spring-boot");
- map.put("SERVER_PORT", "8585");
- application.setDefaultProperties(map);
- application.run(args);

Using EmbeddedServletContainerCustomizer

- Change embedded servlet container default settings by registering a bean that implements **EmbeddedServletContainerCustomizer** interface and override it's **customize()** method.
- **CustomizerBean.java**
- @Component
- public class CustomizerBean implements EmbeddedServletContainerCustomizer {
- 	@Override
- 	public void customize(ConfigurableEmbeddedServletContainer container) {
- 		container.setContextPath("/spring-boot-app");
- 		container.setPort(8585);
- 	}
- }

Introduction to Spring JDBC

- **The JdbcTemplate**
  - Spring provides a nice abstraction on top of JDBC API using JdbcTemplate , which is part of **org.springframework.jdbc.core.JdbcTemplate**.
  - It is the central class which takes care of creation and release of resources such as creating and closing of connection object etc.
  - It internally uses JDBC API, but eliminates a lot of problems of JDBC API.

- **Commonly used method JdbcTemplate**
  - **public int update(String query,Object... args)** : Used to insert, update and delete records using PreparedStatement using given arguments
  - **public List query(String sql, RowMapper rse)** : Used to fetch records using RowMapper.

RowMapper Interface

- **RowMapper Interface**
  - Used to fetch the records from the database using query() method of JdbcTemplate class.
  - it maps a row of the relations with the instance of user-defined class.
  - It iterates the ResultSet internally and adds it into the collection.
- **Syntax**
- public T query(String sql,RowMapper<T> rm)

RowMapper Interface(contd..)

- **Method of RowMapper interface**
  - It defines only one method mapRow that accepts ResultSet instance and int as the parameter list.

- **Syntax**
- public T mapRow(ResultSet rs, int rowNumber)throws SQLException
  - rs - the ResultSet to map (pre-initialized for the current row)
  - rowNum - the number of the current row

Spring Boot JDBC

- By using Spring Boot we can auto configure JdbcTemplate beans, by adding **spring-boot-starter-jdbc module**.
- **pom.xml**
- <dependency>
- <groupId>org.springframework.boot</groupId>
- <artifactId>spring-boot-starter-jdbc</artifactId>
- </dependency>
- **By adding spring-boot-starter-jdbc module, we get the following auto configuration**
  - The spring-boot-starter-jdbc module transitively pulls tomcat-jdbc-{version}.jar which is used to configure the DataSource bean.
  - PlatformTransactionManager (DataSourceTransactionManager)
  - JdbcTemplate
  - NamedParameterJdbcTemplate

Configure Data Source

- DataSource can be configured in **application.properties** file using prefix spring.datasource.
- Spring boot uses **javax.sql.DataSource** interface to configure DataSource.
- **application.properties**
  - spring.datasource.url=jdbc:oracle:thin:@127.0.0.1:1521:XE
  - spring.datasource.username=system
  - spring.datasource.password=hr
  - spring.datasource.driver-class-name=oracle.jdbc.driver.OracleDriver

Autowiring JdbcTemplate

- **JdbcTemplate**
  - This class can autowired in the classes annotated with spring stereotypes such as @Component, @Service, @Repository and @Controller.
- @Repository
- public class ApplicationDAO {
- @Autowired
- private JdbcTemplate jdbcTemplate;

- }

Implementing RowMapper

- **RowMapper**
  - Create a sub class which implements RowMapper interface and implement it's mapRow() method.

```
public class ProjectRowMapper implements RowMapper<Project> {
        @Override
    public Project mapRow(ResultSet rs, int rownum) throws SQLException {
                    Project p=new Project();
                     p.setId(rs.getInt(1));
                    p.setName(rs.getString(2));
                    p.setDuration(rs.getInt(3));
                   return p;

        }

}
```

Spring Boot Internationalization

- **Internationalization**
  - It is the process of designing a software application so that it can be adapted to various languages and regions without engineering changes.
- It is also abbreviated as i18n (where 18 stands for the number of letters between the first i and last n in internationalization) due to the length of the words.

- **For Internationalization you needs to register following beans**
  - ReloadableResourceBundleMessageSource
  - LocaleResolver
  - LocaleChangeInterceptor

Configure i18n

- **For Maven Project**
- src/
- |-- main/
- |-- resources/
- |-- messages_en.properties
- |-- messages_de.properties
- |-- messages_xx.properties

- Spring Boot application by default will look for internationalization key and values under /src/main/resources folder.

Configuring a ReloadableResourceBundleMessageSource

- **ReloadableResourceBundleMessageSource**
  - It is implementation of MessageSource interface that resolves messages from resource bundles for different locales.

```
@Bean
public ReloadableResourceBundleMessageSource messageSource() {
    ReloadableResourceBundleMessageSource messageSource = new ReloadableResourceBundleMessageSource();
    messageSource.setBasename("classpath:messages");
    return messageSource;
}
```

Configuring a LocaleResolver

- **LocaleResolver**
  - Required to correctly determine which local is currently being used. LocalResolver interface has different implementations based on a request, session, cookies, etc.
  - Allows Spring to know which message file's values to read by determining which locale the application is currently running in.

```
@Bean
public LocaleResolver localeResolver() {
    CookieLocaleResolver clr = new CookieLocaleResolver();
     clr.setDefaultLocale(Locale.US);
    return clr;
}
```

Configuring a LocaleChangeInterceptor

- **LocaleChangeInterceptor**
  - It allows for changing the current locale on every request, using a configurable request parameter
  - It is responsible for swapping out the current locale.
- @Bean
- public LocaleChangeInterceptor localeChangeInterceptor() {
- LocaleChangeInterceptor lci = new LocaleChangeInterceptor();
- lci.setParamName("lang");
- return lci;
- }
- This interceptor will look for a request parameter named ‘lang’ and will use its value to determine which locale to switch to.

Registering Interceptor

- **Registering Interceptor**
  - For any interceptor to take effect, we need to add it to the application's interceptor registry.
  - To register this bean with Spring Boot, we need to override addInterceptor() method in our Configuration class.
  - In order to do that, extends your Configuration class by WebMvcConfigurerAdapter.

```
@Configuration
public class BeanConfig extends WebMvcConfigurerAdapter{
 @Bean
@Override
  public void addInterceptors(InterceptorRegistry registry) {
     registry.addInterceptor(localeChangeInterceptor());
  }
}
```

Spring Boot – Deploy WAR file to Tomcat

- **Steps to be performed**
  - Update packaging to war
  - Mark the embedded servlet container as provided.
  - Extends SpringBootServletInitializer

- **Update Packaging to WAR**
- <packaging>war</packaging>
- **pom.xml**
- **<!-- marked the embedded servlet container as provided -->**
  - <dependency>
  - <groupId>org.springframework.boot</groupId>
  - <artifactId>spring-boot-starter-tomcat</artifactId>
  - <scope>provided</scope>
  - </dependency>
- This is required to avoid conflict between the embedded container and the Tomcat server.

Spring Boot – Deploy WAR file to Tomcat(contd..)

- **Modify @SpringBootApplication class**
  - Extend the class from SpringBootServletInitializer and override the configure() method

```
@SpringBootApplication
public class SpringApplication extends SpringBootServletInitializer{
    @Override
    protected SpringApplicationBuilder configure(SpringApplicationBuilder application) {
    return application.sources(SpringApplication.class);
        }
public static void main(String[] args) {
SpringApplication.run(SpringApplication.class, args);
System.out.println("Application is Running");
    }
}
```

Spring Boot – Deploy WAR file to Tomcat(contd..)

- **SpringBootServletInitializer Class**
  - public abstract class SpringBootServletInitializer extends Object implements org.springframework.web.WebApplicationInitializer .It add a web entry point into your application
  - To configure the application override the configure(SpringApplicationBuilder) method (calling SpringApplicationBuilder.sources(Class<?>... sources)
  - This class makes use of Spring Framework's Servlet 3.0 support and allows you to configure your application when it's launched by the servlet container without using web.xml.
  - it Binds Servlet, Filter and ServletContextInitializer beans from the application context to the servlet container.

infogain

Thank You