13.6 Modeling JDBC operations as Java objects

The `org.springframework.jdbc.object` package contains classes that allow one to access the database in a more object-oriented manner. By way of an example, one can execute queries and get the results back as a list containing business objects with the relational column data mapped to the properties of the business object. One can also execute stored procedures and run update, delete and insert statements.

Note

There is a view borne from experience acquired in the field amongst some of the Spring developers that the various RDBMS operation classes described below (with the exception of the `StoredProcedure` class) can often be replaced with straight `JdbcTemplate` calls... often it is simpler to use and plain easier to read a DAO method that simply calls a method on a `JdbcTemplate` direct (as opposed to encapsulating a query as a full-blown class).

It must be stressed however that this is just a *view*... if you feel that you are getting measurable value from using the RDBMS operation classes, feel free to continue using these classes.

13.6.1 `SqlQuery`

`SqlQuery` is a reusable, threadsafe class that encapsulates an SQL query. Subclasses must implement the `newRowMapper(..)` method to provide a `RowMapper` instance that can create one object per row obtained from iterating over the `ResultSet` that is created during the execution of the query. The `SqlQuery` class is rarely used directly since the `MappingSqlQuery` subclass provides a much more convenient implementation for mapping rows to Java classes. Other implementations that extend `SqlQuery` are `MappingSqlQueryWithParameters` and `UpdatableSqlQuery`.

13.6.2 `MappingSqlQuery`

`MappingSqlQuery` is a reusable query in which concrete subclasses must implement the abstract `mapRow(..)` method to convert each row of the supplied `ResultSet` into an object. Find below a brief example of a custom query that maps the data from the customer relation to an instance of the `Customer` class.

```
private class CustomerMappingQuery extends MappingSqlQuery {

    public CustomerMappingQuery(DataSource ds) {
        super(ds, "SELECT id, name FROM customer WHERE id = ?");
        super.declareParameter(new SqlParameter("id", Types.INTEGER));
        compile();
    }
    public Object mapRow(ResultSet rs, int rowNumber) throws SQLException {
        Customer cust = new Customer();
        cust.setId((Integer) rs.getObject("id"));
        cust.setName(rs.getString("name"));
        return cust;
    }
}
```

We provide a constructor for this customer query that takes the `DataSource` as the only parameter. In this constructor we call the constructor on the superclass with the `DataSource` and the SQL that should be executed to retrieve the rows for this query. This SQL will be used to create a `PreparedStatement` so it may contain place holders for any parameters to be passed in during execution. Each parameter must be declared using the `declareParameter` method passing in an `SqlParameter`. The `SqlParameter` takes a name and the JDBC type as defined in `java.sql.Types`. After all parameters have been defined we call the `compile()` method so the statement can be prepared and later be executed.

```
public Customer getCustomer(Integer id) {
    CustomerMappingQuery custQry = new CustomerMappingQuery(dataSource);
    Object[] parms = new Object[1];
    parms[0] = id;
    List customers = custQry.execute(parms);
    if (customers.size() > 0) {
        return (Customer) customers.get(0);
    }
    else {
        return null;
    }
}
```

The method in this example retrieves the customer with the id that is passed in as the only parameter. After creating an instance of the `CustomerMappingQuery` class we create an array of objects that will contain all parameters that are passed in. In this case there is only one parameter and it is passed in as an `Integer`. Now we are ready to execute the query using this array of parameters and we get a `List` that contains a `Customer` object for each row that was returned for our query. In this case it will only be one entry if there was a match.

13.6.3 `SqlUpdate`

The `SqlUpdate` class encapsulates an SQL update. Like a query, an update object is reusable, and like all `RdbmsOperation` classes, an update can have parameters and is defined in SQL. This class provides a number of `update(..)` methods analogous to the `execute(..)` methods of query objects. This class is concrete. Although it can be subclassed (for example to add a custom update method) it can easily be parameterized by setting SQL and declaring parameters.

```
import java.sql.Types;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlParameter;
import org.springframework.jdbc.object.SqlUpdate;

public class UpdateCreditRating extends SqlUpdate {

    public UpdateCreditRating(DataSource ds) {
        setDataSource(ds);
        setSql("update customer set credit_rating = ? where id = ?");
        declareParameter(new SqlParameter(Types.NUMERIC));
        declareParameter(new SqlParameter(Types.NUMERIC));
        compile();
    }

    /**
     * @param id for the Customer to be updated
     * @param rating the new value for credit rating
     * @return number of rows updated
     */
    public int run(int id, int rating) {
        Object[] params =
            new Object[] {
                new Integer(rating),
                new Integer(id)};
        return update(params);
    }
}
```

13.6.4 `StoredProcedure`

The `StoredProcedure` class is a superclass for object abstractions of RDBMS stored procedures. This class is `abstract`, and its various `execute(..)` methods have `protected` access, preventing use other than through a subclass that offers tighter typing.

The inherited `sql` property will be the name of the stored procedure in the RDBMS.

To define a parameter to be used for the StoredProcedure classe, you use an `SqlParameter` or one of its subclasses. You must specify the parameter name and SQL type in the constructor. The SQL type is specified using the `java.sql.Types` constants. We have already seen declarations like:

```
   new SqlParameter("in_id", Types.NUMERIC),
   new SqlOutParameter("out_first_name", Types.VARCHAR),
```

The first line with the `SqlParameter` declares an in parameter. In parameters can be used for both stored procedure calls and for queries using the `SqlQuery` and its subclasses covered in the following section.

The second line with the `SqlOutParameter` declares an out parameter to be used in the stored procedure call. There is also an `SqlInOutParameter` for inout parameters, parameters that provide an in value to the procedure and that also return a value

Note

Parameters declared as `SqlParameter` and `SqlInOutParameter` will always be used to provide input values. In addition to this any parameter declared as `SqlOutParameter` where an non-null input value is provided will also be used as an input paraneter.

In addition to the name and the SQL type you can specify additional options. For in parameters you can specify a scale for numeric data or a type name for custom database types. For out parameters you can provide a `RowMapper` to handle mapping of rows returned from a REF cursor. Another option is to specify an `SqlReturnType` that provides and opportunity to define customized handling of the return values.

Here is an example of a program that calls a function, `sysdate()`, that comes with any Oracle database. To use the stored procedure functionality one has to create a class that extends `StoredProcedure`. There are no input parameters, but there is an output parameter that is declared as a date type using the class `SqlOutParameter`. The `execute()` method returns a map with an entry for each declared output parameter using the parameter name as the key.

```
import java.sql.Types;
import java.util.HashMap;
import java.util.Iterator;
import java.util.Map;

import javax.sql.DataSource;

import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.datasource.*;
import org.springframework.jdbc.object.StoredProcedure;

public class TestStoredProcedure {

    public static void main(String[] args)  {
        TestStoredProcedure t = new TestStoredProcedure();
        t.test();
        System.out.println("Done!");
    }

    void test() {
        DriverManagerDataSource ds = new DriverManagerDataSource();
        ds.setDriverClassName("oracle.jdbc.OracleDriver");
        ds.setUrl("jdbc:oracle:thin:@localhost:1521:mydb");
        ds.setUsername("scott");
        ds.setPassword("tiger");

        MyStoredProcedure sproc = new MyStoredProcedure(ds);
        Map results = sproc.execute();
        printMap(results);
    }

    private class MyStoredProcedure extends StoredProcedure {

        private static final String SQL = "sysdate";

        public MyStoredProcedure(DataSource ds) {
            setDataSource(ds);
            setFunction(true);
            setSql(SQL);
            declareParameter(new SqlOutParameter("date", Types.DATE));
            compile();
        }

        public Map execute() {
            // the 'sysdate' sproc has no input parameters, so an empty Map
is supplied...
            return execute(new HashMap());
        }
    }

    private static void printMap(Map results) {
        for (Iterator it = results.entrySet().iterator(); it.hasNext(); ) {
            System.out.println(it.next());
        }
    }
}
```

Find below an example of a `StoredProcedure` that has two output parameters (in this case Oracle REF cursors).

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAndGenresStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "AllTitlesAndGenres";

    public TitlesAndGenresStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR,
new TitleMapper()));
        declareParameter(new SqlOutParameter("genres", OracleTypes.CURSOR,
new GenreMapper()));
```

```
        compile();
    }

    public Map execute() {
        // again, this sproc has no input parameters, so an empty Map is
supplied...
        return super.execute(new HashMap());
    }
}
```

Notice how the overloaded variants of the `declareParameter(..)` method that have been used in the `TitlesAndGenresStoredProcedure` constructor are passed `RowMapper` implementation instances; this is a very convenient and powerful way to reuse existing functionality. (The code for the two `RowMapper` implementations is provided below in the interest of completeness.)

Firstly the `TitleMapper` class, which simply maps a `ResultSet` to a `Title` domain object for each row in the supplied `ResultSet`.

```
import com.foo.sprocs.domain.Title;
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

public final class TitleMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        Title title = new Title();
        title.setId(rs.getLong("id"));
        title.setName(rs.getString("name"));
        return title;
    }
}
```

Secondly, the `GenreMapper` class, which again simply maps a `ResultSet` to a `Genre` domain object for each row in the supplied `ResultSet`.

```
import org.springframework.jdbc.core.RowMapper;

import java.sql.ResultSet;
import java.sql.SQLException;

import com.foo.domain.Genre;

public final class GenreMapper implements RowMapper {

    public Object mapRow(ResultSet rs, int rowNum) throws SQLException {
        return new Genre(rs.getString("name"));
    }
}
```

If one needs to pass parameters to a stored procedure (that is the stored procedure has been declared as having one or more input parameters in its definition in the RDBMS), one would code a strongly typed `execute(..)` method which would delegate to the superclass' (untyped) `execute(Map parameters)` (which has `protected` access); for example:

```
import oracle.jdbc.driver.OracleTypes;
import org.springframework.jdbc.core.SqlOutParameter;
import org.springframework.jdbc.object.StoredProcedure;

import javax.sql.DataSource;
import java.util.HashMap;
import java.util.Map;

public class TitlesAfterDateStoredProcedure extends StoredProcedure {

    private static final String SPROC_NAME = "TitlesAfterDate";
    private static final String CUTOFF_DATE_PARAM = "cutoffDate";

    public TitlesAfterDateStoredProcedure(DataSource dataSource) {
        super(dataSource, SPROC_NAME);
        declareParameter(new SqlParameter(CUTOFF_DATE_PARAM, Types.DATE);
        declareParameter(new SqlOutParameter("titles", OracleTypes.CURSOR,
new TitleMapper()));
        compile();
    }

    public Map execute(Date cutoffDate) {
        Map inputs = new HashMap();
        inputs.put(CUTOFF_DATE_PARAM, cutoffDate);
        return super.execute(inputs);
    }
}
```

13.6.5 `SqlFunction`

The `SqlFunction` RDBMS operation class encapsulates an SQL "function" wrapper for a query that returns a single row of results. The default behavior is to return an `int`, but that can be overridden by using the methods with an extra return type parameter. This is similar to using the `queryForXxx` methods of the `JdbcTemplate`. The advantage with `SqlFunction` is that you don't have to create the `JdbcTemplate`, it is done behind the scenes.

This class is intended to use to call SQL functions that return a single result using a query like "select user()" or "select sysdate from dual". It is not intended for calling more complex stored functions or for using a `CallableStatement` to invoke a stored procedure or stored function. (Use the `StoredProcedure` or `SqlCall` classes for this type of processing).

`SqlFunction` is a concrete class, and there is typically no need to subclass it. Code using this package can create an object of this type, declaring SQL and parameters, and then invoke the appropriate run method repeatedly to execute the function. Here is an example of retrieving the count of rows from a table:

```
public int countRows() {
    SqlFunction sf = new SqlFunction(dataSource, "select count(*) from
mytable");
    sf.compile();
    return sf.run();
}
```

Spring Boot - Database Handling

Spring Boot provides a very good support to create a DataSource for Database. We need not write any extra code to create a DataSource in Spring Boot. Just adding the dependencies and doing the configuration details is enough to create a DataSource and connect the Database.

In this chapter, we are going to use Spring Boot JDBC driver connection to connect the database.

First, we need to add the Spring Boot Starter JDBC dependency in our build configuration file.

Maven users can add the following dependencies in the pom.xml file.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-jdbc</artifactId>
</dependency>
```

Gradle users can add the following dependencies in the build.gradle file.

compile('org.springframework.boot:spring-boot-starter-jdbc')

Connect to H2 database

To connect the H2 database, we need to add the H2 database dependency in our build configuration file.

For Maven users, add the below dependency in your pom.xml file.

```
<dependency>
```

```
  <groupId>com.h2database</groupId>
  <artifactId>h2</artifactId>
</dependency>
```

For Gradle users, add the below dependency in your build.gradle file.

```
compile('com.h2database:h2')
```

We need to create the schema.sql file and data.sql file under the classpath src/main/resources directory to connect the H2 database.

The schema.sql file is given below.

```
CREATE TABLE PRODUCT (ID INT PRIMARY KEY, PRODUCT_NAME VARCHAR(25));
```

The data.sql file is given below.

```
INSERT INTO PRODUCT (ID,PRODUCT_NAME) VALUES (1,'Honey');
INSERT INTO PRODUCT (ID,PRODUCT_NAME) VALUES (2,'Almond');
```

Connect MySQL

To connect the MySQL database, we need to add the MySQL dependency into our build configuration file.

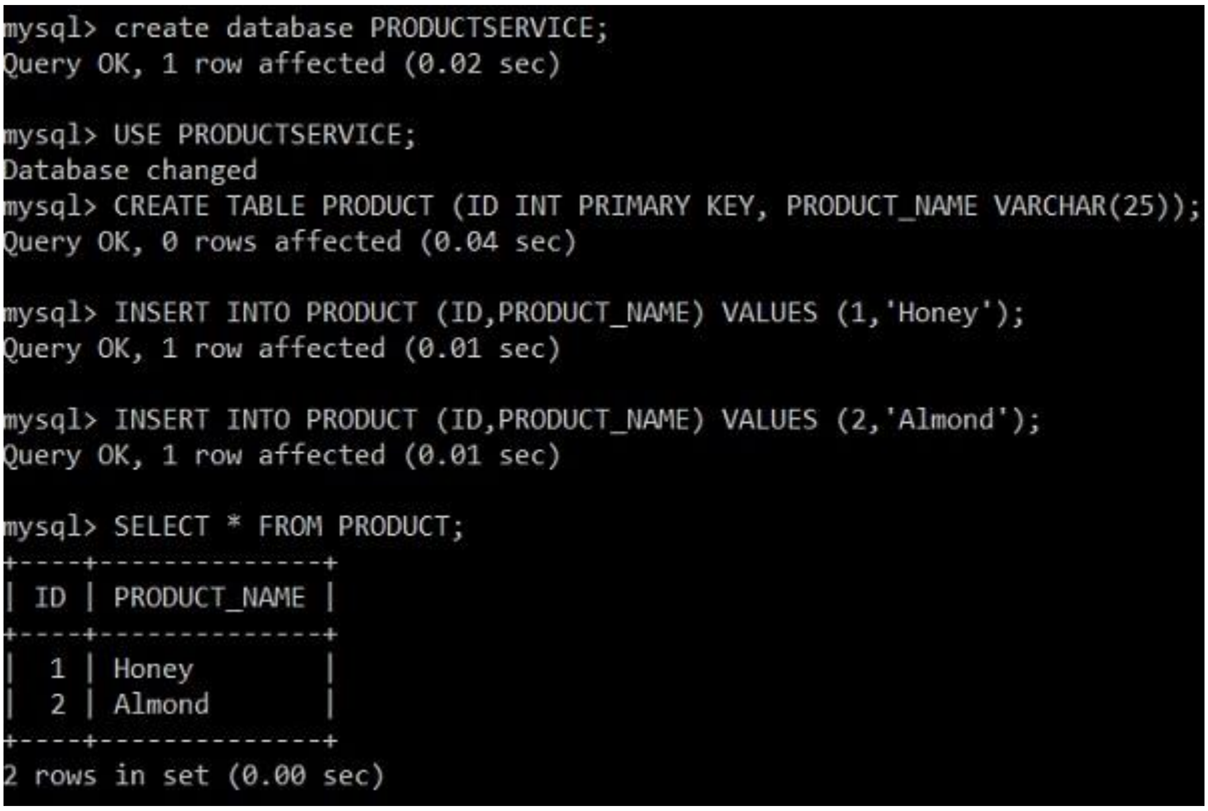
For Maven users, add the following dependency in your pom.xml file.

```
<dependency>
  <groupId>mysql</groupId>
  <artifactId>mysql-connector-java</artifactId>
</dependency>
```

For Gradle users, add the following dependency in your build.gradle file.

```
compile('mysql:mysql-connector-java')
```

Now, create database and tables in MySQL as shown –



```
mysql> create database PRODUCTSERVICE;
Query OK, 1 row affected (0.02 sec)

mysql> USE PRODUCTSERVICE;
Database changed
mysql> CREATE TABLE PRODUCT (ID INT PRIMARY KEY, PRODUCT_NAME VARCHAR(25));
Query OK, 0 rows affected (0.04 sec)

mysql> INSERT INTO PRODUCT (ID,PRODUCT_NAME) VALUES (1,'Honey');
Query OK, 1 row affected (0.01 sec)

mysql> INSERT INTO PRODUCT (ID,PRODUCT_NAME) VALUES (2,'Almond');
Query OK, 1 row affected (0.01 sec)

mysql> SELECT * FROM PRODUCT;
+----+--------------+
| ID | PRODUCT_NAME |
+----+--------------+
|  1 | Honey        |
|  2 | Almond       |
+----+--------------+
2 rows in set (0.00 sec)
```

For properties file users, add the following properties in the application.properties file.

```
spring.datasource.driverClassName = com.mysql.jdbc.Driver
spring.datasource.url = jdbc:mysql://localhost:3306/PRODUCTSERVICE?autoreconnect = true
spring.datasource.username = root
spring.datasource.password = root
spring.datasource.testOnBorrow = true
spring.datasource.testWhileIdle = true
spring.datasource.timeBetweenEvictionRunsMillis = 60000
spring.datasource.minEvictableIdleTimeMillis = 30000
spring.datasource.validationQuery = SELECT 1
spring.datasource.max-active = 15
spring.datasource.max-idle = 10
spring.datasource.max-wait = 8000
```

For YAML users, add the following properties in the application.yml file.

```
spring:
  datasource:
    driverClassName: com.mysql.jdbc.Driver
    url: "jdbc:mysql://localhost:3306/PRODUCTSERVICE?autoreconnect=true"
    username: "root"
    password: "root"
    testOnBorrow: true
    testWhileIdle: true
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 30000
    validationQuery: SELECT 1
    max-active: 15
    max-idle: 10
    max-wait: 8000
```

Connect Redis

Redis is an open source database used to store the in-memory data structure. To connect the Redis database in Spring Boot application, we need to add the Redis dependency in our build configuration file.

Maven users should add the following dependency in your pom.xml file.

```
<dependency>
  <groupId>org.springframework.boot</groupId>
  <artifactId>spring-boot-starter-redis</artifactId>
</dependency>
```

Gradle users should add the following dependency in your build.gradle file.

compile('org.springframework.boot:spring-boot-starter-data-redis')

For Redis connection, we need to use RedisTemplate. For RedisTemplate we need to provide the JedisConnectionFactory details.

```
@Bean
JedisConnectionFactory jedisConnectionFactory() {
  JedisConnectionFactory jedisConFactory = new JedisConnectionFactory();
  jedisConFactory.setHostName("localhost");
  jedisConFactory.setPort(6000);
  jedisConFactory.setUsePool(true);
  return jedisConFactory;
}
@Bean
public RedisTemplate<String, Object> redisTemplate() {
  RedisTemplate<String, Object> template = new RedisTemplate<>();
  template.setConnectionFactory(jedisConnectionFactory());
  template.setKeySerializer(new StringRedisSerializer());
  template.setHashKeySerializer(new StringRedisSerializer());
  template.setHashValueSerializer(new StringRedisSerializer());
  template.setValueSerializer(new StringRedisSerializer());
  return template;
}
```

Now auto wire the RedisTemplate class and access the data from Redis database.

```
@Autowired

RedisTemplate<String, Object> redis;
Map<Object,Object> datalist = redis.opsForHash().entries(“Redis_code_index_key”);
```

JDBCTemplate

To access the Relational Database by using JdbcTemplate in Spring Boot application, we need to add the Spring Boot Starter JDBC dependency in our build configuration file.

Then, if you @Autowired the JdbcTemplate class, Spring Boot automatically connects the Database and sets the Datasource for the JdbcTemplate object.

```
@Autowired
JdbcTemplate jdbcTemplate;
Collection<Map<String, Object>> rows = jdbc.queryForList("SELECT QUERY");
```

The @Repository annotation should be added into the class file. The @Repository annotation is used to create database repository for your Spring Boot application.

```
@Repository
public class ProductServiceDAO {
}
```

Multiple DataSource

We can keep 'n' number Datasources in a single Spring Boot application. The example given here shows how to create more than 1 data source in Spring Boot application. Now, add the two data source configuration details in the application properties file.

For properties file users, add the following properties into your application.properties file.

```
spring.dbProductService.driverClassName = com.mysql.jdbc.Driver
spring.dbProductService.url = jdbc:mysql://localhost:3306/PRODUCTSERVICE?autoreconnect = true
spring.dbProductService.username = root
spring.dbProductService.password = root
spring.dbProductService.testOnBorrow = true
spring.dbProductService.testWhileIdle = true
spring.dbProductService.timeBetweenEvictionRunsMillis = 60000
spring.dbProductService.minEvictableIdleTimeMillis = 30000
spring.dbProductService.validationQuery = SELECT 1
spring.dbProductService.max-active = 15
spring.dbProductService.max-idle = 10
spring.dbProductService.max-wait = 8000

spring.dbUserService.driverClassName = com.mysql.jdbc.Driver
spring.dbUserService.url = jdbc:mysql://localhost:3306/USERSERVICE?autoreconnect = true
spring.dbUserService.username = root
spring.dbUserService.password = root
spring.dbUserService.testOnBorrow = true
spring.dbUserService.testWhileIdle = true
spring.dbUserService.timeBetweenEvictionRunsMillis = 60000
spring.dbUserService.minEvictableIdleTimeMillis = 30000
spring.dbUserService.validationQuery = SELECT 1
spring.dbUserService.max-active = 15
spring.dbUserService.max-idle = 10
spring.dbUserService.max-wait = 8000
```

Yaml users should add the following properties in your application.yml file.

```
spring:
  dbProductService:
    driverClassName: com.mysql.jdbc.Driver
    url: "jdbc:mysql://localhost:3306/PRODUCTSERVICE?autoreconnect=true"
    password: "root"
    username: "root"
    testOnBorrow: true
    testWhileIdle: true
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 30000
    validationQuery: SELECT 1
    max-active: 15
    max-idle: 10
    max-wait: 8000
  dbUserService:
    driverClassName: com.mysql.jdbc.Driver
    url: "jdbc:mysql://localhost:3306/USERSERVICE?autoreconnect=true"
    password: "root"
    username: "root"
    testOnBorrow: true
    testWhileIdle: true
    timeBetweenEvictionRunsMillis: 60000
    minEvictableIdleTimeMillis: 30000
    validationQuery: SELECT 1
    max-active: 15
    max-idle: 10
    max-wait: 8000
```

Now, create a Configuration class to create a DataSource and JdbcTemplate for multiple data sources.

```
import javax.sql.DataSource;

import org.springframework.beans.factory.annotation.Autowired;
import org.springframework.beans.factory.annotation.Qualifier;
import org.springframework.boot.autoconfigure.jdbc.DataSourceBuilder;
import org.springframework.boot.context.properties.ConfigurationProperties;
import org.springframework.context.annotation.Bean;
import org.springframework.context.annotation.Configuration;
import org.springframework.context.annotation.Primary;
import org.springframework.jdbc.core.JdbcTemplate;

@Configuration
public class DatabaseConfig {
  @Bean(name = "dbProductService")
  @ConfigurationProperties(prefix = "spring.dbProductService")
  @Primary
  public DataSource createProductServiceDataSource() {
    return DataSourceBuilder.create().build();
  }
  @Bean(name = "dbUserService")
  @ConfigurationProperties(prefix = "spring.dbUserService")
  public DataSource createUserServiceDataSource() {
    return DataSourceBuilder.create().build();
  }
  @Bean(name = "jdbcProductService")
  @Autowired
  public JdbcTemplate createJdbcTemplate_ProductService(@Qualifier("dbProductService") DataSource
productServiceDS) {
    return new JdbcTemplate(productServiceDS);
  }
  @Bean(name = "jdbcUserService")
  @Autowired
  public JdbcTemplate createJdbcTemplate_UserService(@Qualifier("dbUserService") DataSource
userServiceDS) {
    return new JdbcTemplate(userServiceDS);
  }
}
```

Then, auto wire the JDBCTemplate object by using @Qualifier annotation.

```
@Qualifier("jdbcProductService")
@Autowired
JdbcTemplate jdbcTemplate;
```

```
@Qualifier("jdbcUserService")
@Autowired
JdbcTemplate jdbcTemplate;
```

Developing Web Applications

Spring Boot is well suited for web application development. You can create a self-contained HTTP server by using embedded Tomcat, Jetty, Undertow, or Netty. Most web applications use the `spring-boot-starter-web` module to get up and running quickly. You can also choose to build reactive web applications by using the `spring-boot-starter-webflux` module.

If you have not yet developed a Spring Boot web application, you can follow the "Hello World!" example in the *Getting started* section.

29.1 The “Spring Web MVC Framework”

The Spring Web MVC framework (often referred to as simply “Spring MVC”) is a rich “model view controller” web framework. Spring MVC lets you create special `@Controller` or `@RestController` beans to handle incoming HTTP requests. Methods in your controller are mapped to HTTP by using `@RequestMapping` annotations.

The following code shows a typical `@RestController` that serves JSON data:

```
@RestController
@RequestMapping(value="/users")
public class MyRestController {

	@RequestMapping(value="/\{user}", method=RequestMethod.GET)
	public User getUser(@PathVariable Long user) {
		// ...
	}

	@RequestMapping(value="/\{user}/customers",
method=RequestMethod.GET)
	List<Customer> getUserCustomers(@PathVariable Long user) {
		// ...
	}

	@RequestMapping(value="/\{user}", method=RequestMethod.DELETE)
```

```
        public User deleteUser(@PathVariable Long user) {
                // ...
        }

}
```

Spring MVC is part of the core Spring Framework, and detailed information is available in the reference documentation. There are also several guides that cover Spring MVC available at spring.io/guides.

29.1.1 Spring MVC Auto-configuration

Spring Boot provides auto-configuration for Spring MVC that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Inclusion of `ContentNegotiatingViewResolver` and `BeanNameViewResolver` beans.
- Support for serving static resources, including support for WebJars (covered later in this document)).
- Automatic registration of `Converter`, `GenericConverter`, and `Formatter` beans.
- Support for `HttpMessageConverters` (covered later in this document).
- Automatic registration of `MessageCodesResolver` (covered later in this document).
- Static `index.html` support.
- Custom `Favicon` support (covered later in this document).
- Automatic use of a `ConfigurableWebBindingInitializer` bean (covered later in this document).

If you want to keep Spring Boot MVC features and you want to add additional MVC configuration (interceptors, formatters, view controllers, and other features), you can add your own `@Configuration` class of type `WebMvcConfigurer` but **without** `@EnableWebMvc`. If you wish to provide custom instances of `RequestMappingHandlerMapping`, `RequestMappingHandlerAdapter`, or `ExceptionHandlerExceptionResolver`, you can declare a `WebMvcRegistrationsAdapter` instance to provide such components.

If you want to take complete control of Spring MVC, you can add your own `@Configuration` annotated with `@EnableWebMvc`.

29.1.2 HttpMessageConverters

Spring MVC uses the `HttpMessageConverter` interface to convert HTTP requests and responses. Sensible defaults are included out of the box. For example, objects can be automatically converted to JSON (by using the Jackson library) or XML (by using the Jackson XML extension, if available, or by using JAXB if the Jackson XML extension is not available). By default, strings are encoded in `UTF-8`.

If you need to add or customize converters, you can use Spring Boot's `HttpMessageConverters` class, as shown in the following listing:

```java
import
org.springframework.boot.autoconfigure.http.HttpMessageConverters;
import org.springframework.context.annotation.*;
import org.springframework.http.converter.*;

@Configuration
public class MyConfiguration {

	@Bean
	public HttpMessageConverters customConverters() {
		HttpMessageConverter<?> additional = ...
		HttpMessageConverter<?> another = ...
		return new HttpMessageConverters(additional, another);
	}

}
```

Any `HttpMessageConverter` bean that is present in the context is added to the list of converters. You can also override default converters in the same way.

29.1.3 Custom JSON Serializers and Deserializers

If you use Jackson to serialize and deserialize JSON data, you might want to write your own `JsonSerializer` and `JsonDeserializer` classes. Custom serializers are usually registered with Jackson through a module, but Spring Boot provides an alternative `@JsonComponent` annotation that makes it easier to directly register Spring Beans.

You can use the `@JsonComponent` annotation directly on `JsonSerializer` or `JsonDeserializer` implementations. You can also use it on classes that contain serializers/deserializers as inner classes, as shown in the following example:

```java
import java.io.*;
import com.fasterxml.jackson.core.*;
import com.fasterxml.jackson.databind.*;
import org.springframework.boot.jackson.*;

@JsonComponent
public class Example {

	public static class Serializer extends
JsonSerializer<SomeObject> {
		// ...
	}

	public static class Deserializer extends
JsonDeserializer<SomeObject> {
		// ...
	}

}
```

All `@JsonComponent` beans in the `ApplicationContext` are automatically registered with Jackson. Because `@JsonComponent` is meta-annotated with `@Component`, the usual component-scanning rules apply.

Spring Boot also provides `JsonObjectSerializer` and `JsonObjectDeserializer` base classes that provide useful alternatives to the standard Jackson versions when serializing objects. See `JsonObjectSerializer` and `JsonObjectDeserializer` in the Javadoc for details.

29.1.4 MessageCodesResolver

Spring MVC has a strategy for generating error codes for rendering error messages from binding errors: `MessageCodesResolver`. If you set the `spring.mvc.message-codes-resolver.format` property `PREFIX_ERROR_CODE` or `POSTFIX_ERROR_CODE`, Spring Boot creates one for you (see the enumeration in `DefaultMessageCodesResolver.Format`).

29.1.5 Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath or from the root of the `ServletContext`. It uses the `ResourceHttpRequestHandler` from Spring MVC so that you can modify that behavior by adding your own `WebMvcConfigurer` and overriding the `addResourceHandlers` method.

In a stand-alone web application, the default servlet from the container is also enabled and acts as a fallback, serving content from the root of the `ServletContext` if Spring decides not to handle it. Most of the time, this does not happen (unless you modify the default MVC configuration), because Spring can always handle requests through the `DispatcherServlet`.

By default, resources are mapped on `/**`, but you can tune that with the `spring.mvc.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.mvc.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using the `spring.resources.static-locations` property (replacing the default values with a list of directory locations). The root Servlet context path, `"/"`, is automatically added as a location as well.

In addition to the “standard” static resource locations mentioned earlier, a special case is made for Webjars content. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

Do not use the `src/main/webapp` directory if your application is packaged as a jar. Although this directory is a common standard, it works **only** with war packaging, and it is silently ignored by most build tools if you generate a jar.

Spring Boot also supports the advanced resource handling features provided by Spring MVC, allowing use cases such as cache-busting static resources or using version agnostic URLs for Webjars.

To use version agnostic URLs for Webjars, add the `webjars-locator-core` dependency. Then declare your Webjar. Using jQuery as an example, adding `"/webjars/jquery/jquery.min.js"` results in `"/webjars/jquery/x.y.z/jquery.min.js"`. where `x.y.z` is the Webjar version.

If you use JBoss, you need to declare the `webjars-locator-jboss-vfs` dependency instead of the `webjars-locator-core`. Otherwise, all Webjars resolve as a `404`.

To use cache busting, the following configuration configures a cache busting solution for all static resources, effectively adding a content hash, such as `<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`, in URLs:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
```

Links to resources are rewritten in templates at runtime, thanks to a `ResourceUrlEncodingFilter` that is auto-configured for Thymeleaf and FreeMarker. You should manually declare this filter when using JSPs. Other template engines are currently not automatically supported but can be with custom template macros/helpers and the use of the `ResourceUrlProvider`.

When loading resources dynamically with, for example, a JavaScript module loader, renaming files is not an option. That is why other strategies are also supported and can be combined. A "fixed" strategy adds a static version string in the URL without changing the file name, as shown in the following example:

```
spring.resources.chain.strategy.content.enabled=true
spring.resources.chain.strategy.content.paths=/**
spring.resources.chain.strategy.fixed.enabled=true
spring.resources.chain.strategy.fixed.paths=/js/lib/
spring.resources.chain.strategy.fixed.version=v12
```

With this configuration, JavaScript modules located under `"/js/lib/"` use a fixed versioning strategy (`"/v12/js/lib/mymodule.js"`), while other resources still use the content one (`<link href="/css/spring-2a2d595e6ed9a0b24f027f2b63b134d6.css"/>`).

See `ResourceProperties` for more supported options.

This feature has been thoroughly described in a dedicated blog post and in Spring Framework's reference documentation.

29.1.6 Welcome Page

Spring Boot supports both static and templated welcome pages. It first looks for an `index.html` file in the configured static content locations. If one is not found, it then looks for an `index` template. If either is found, it is automatically used as the welcome page of the application.

29.1.7 Custom Favicon

Spring Boot looks for a `favicon.ico` in the configured static content locations and the root of the classpath (in that order). If such a file is present, it is automatically used as the favicon of the application.

29.1.8 Path Matching and Content Negotiation

Spring MVC can map incoming HTTP requests to handlers by looking at the request path and matching it to the mappings defined in your application (for example, `@GetMapping` annotations on Controller methods).

Spring Boot chooses to disable suffix pattern matching by default, which means that requests like `"GET /projects/spring-boot.json"` won't be matched to `@GetMapping("/projects/spring-boot")` mappings. This is considered as a best practice for Spring MVC applications. This feature was mainly useful in the past for HTTP clients which did not send proper "Accept" request headers; we needed to make sure to send the correct Content Type to the client. Nowadays, Content Negotiation is much more reliable.

There are other ways to deal with HTTP clients that don't consistently send proper "Accept" request headers. Instead of using suffix matching, we can use a query parameter to ensure that requests like `"GET /projects/spring-boot?format=json"` will be mapped to `@GetMapping("/projects/spring-boot")`:

```
spring.mvc.contentnegotiation.favor-parameter=true

# We can change the parameter name, which is "format" by default:
# spring.mvc.contentnegotiation.parameter-name=myparam

# We can also register additional file extensions/media types with:
spring.mvc.contentnegotiation.media-types.markdown=text/markdown
```

If you understand the caveats and would still like your application to use suffix pattern matching, the following configuration is required:

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-suffix-pattern=true
```

Alternatively, rather than open all suffix patterns, it's more secure to just support registered suffix patterns:

```
spring.mvc.contentnegotiation.favor-path-extension=true
spring.mvc.pathmatch.use-registered-suffix-pattern=true

# You can also register additional file extensions/media types with:
# spring.mvc.contentnegotiation.media-types.adoc=text/asciidoc
```

29.1.9 ConfigurableWebBindingInitializer

Spring MVC uses a `WebBindingInitializer` to initialize a `WebDataBinder` for a particular request. If you create your own `ConfigurableWebBindingInitializer` `@Bean`, Spring Boot automatically configures Spring MVC to use it.

29.1.10 Template Engines

As well as REST web services, you can also use Spring MVC to serve dynamic HTML content. Spring MVC supports a variety of templating technologies, including Thymeleaf, FreeMarker, and JSPs. Also, many other templating engines include their own Spring MVC integrations.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Groovy
- Thymeleaf
- Mustache

If possible, JSPs should be avoided. There are several known limitations when using them with embedded servlet containers.

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

Depending on how you run your application, IntelliJ IDEA orders the classpath differently. Running your application in the IDE from its main method results in a different ordering than when you run your application by using Maven or Gradle or from its packaged jar. This can cause Spring Boot to fail to find the templates on the classpath. If you have this problem, you can reorder the classpath in the IDE to place the module's classes and resources first. Alternatively, you can configure the template prefix to search every `templates` directory on the classpath, as follows: `classpath*:/templates/`.

29.1.11 Error Handling

By default, Spring Boot provides an `/error` mapping that handles all errors in a sensible way, and it is registered as a "global" error page in the servlet container. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a "whitelabel" error view that renders the same data in HTML format (to customize it, add a `View` that resolves to `error`). To replace the default behavior completely, you can implement `ErrorController` and register a bean definition of that type or add a bean of type `ErrorAttributes` to use the existing mechanism but replace the contents.

The `BasicErrorController` can be used as a base class for a custom `ErrorController`. This is particularly useful if you want to add a handler for a new content type (the default is to handle `text/html` specifically and provide a fallback for everything else). To do so, extend `BasicErrorController`, add a public method with a `@RequestMapping` that has a `produces` attribute, and create a bean of your new type.

You can also define a class annotated with `@ControllerAdvice` to customize the JSON document to return for a particular controller and/or exception type, as shown in the following example:

```
@ControllerAdvice(basePackageClasses = AcmeController.class)
public class AcmeControllerAdvice extends
ResponseEntityExceptionHandler {

	@ExceptionHandler(YourException.class)
	@ResponseBody
	ResponseEntity<?> handleControllerException(HttpServletRequest
request, Throwable ex) {
		HttpStatus status = getStatus(request);
		return new ResponseEntity<>(new
CustomErrorType(status.value(), ex.getMessage()), status);
	}

	private HttpStatus getStatus(HttpServletRequest request) {
```

```
                Integer statusCode = (Integer)
request.getAttribute("javax.servlet.error.status_code");
                if (statusCode == null) {
                        return HttpStatus.INTERNAL_SERVER_ERROR;
                }
                return HttpStatus.valueOf(statusCode);
        }

}
```

In the preceding example, if `YourException` is thrown by a controller defined in the same package as `AcmeController`, a JSON representation of the `CustomErrorType` POJO is used instead of the `ErrorAttributes` representation.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or be built by using templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a FreeMarker template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.ftl
             +- <other templates>
```

For more complex mappings, you can also add beans that implement the `ErrorViewResolver` interface, as shown in the following example:

```
public class MyErrorViewResolver implements ErrorViewResolver {

	@Override
	public ModelAndView resolveErrorView(HttpServletRequest request,
			HttpStatus status, Map<String, Object> model) {
		// Use the request or status to optionally return a
ModelAndView
		return ...
	}

}
```

You can also use regular Spring MVC features such as `@ExceptionHandler` methods and `@ControllerAdvice`. The `ErrorController` then picks up any unhandled exceptions.

Mapping Error Pages outside of Spring MVC

For applications that do not use Spring MVC, you can use the `ErrorPageRegistrar` interface to directly register `ErrorPages`. This abstraction works directly with the underlying embedded servlet container and works even if you do not have a Spring MVC `DispatcherServlet`.

```
@Bean
public ErrorPageRegistrar errorPageRegistrar(){
	return new MyErrorPageRegistrar();
}

// ...

private static class MyErrorPageRegistrar implements ErrorPageRegistrar
{

	@Override
	public void registerErrorPages(ErrorPageRegistry registry) {
		registry.addErrorPages(new
ErrorPage(HttpStatus.BAD_REQUEST, "/400"));
	}

}
```

If you register an `ErrorPage` with a path that ends up being handled by a `Filter` (as is common with some non-Spring web frameworks, like Jersey and Wicket), then the `Filter` has to be explicitly registered as an `ERROR` dispatcher, as shown in the following example:

```
@Bean
public FilterRegistrationBean myFilter() {
	FilterRegistrationBean registration = new
FilterRegistrationBean();
	registration.setFilter(new MyFilter());
	...
	registration.setDispatcherTypes(EnumSet.allOf(DispatcherType.cla
ss));
	return registration;
}
```

Note that the default `FilterRegistrationBean` does not include the `ERROR` dispatcher type.

CAUTION:When deployed to a servlet container, Spring Boot uses its error page filter to forward a request with an error status to the appropriate error page. The request can only be forwarded to the correct error page if the response has not already been committed. By default, WebSphere Application Server 8.0 and later commits the response upon successful completion of a servlet's service method. You should disable this behavior by setting `com.ibm.ws.webcontainer.invokeFlushAfterService` to `false`.

29.1.12 Spring HATEOAS

If you develop a RESTful API that makes use of hypermedia, Spring Boot provides auto-configuration for Spring HATEOAS that works well with most applications. The auto-configuration replaces the need to use `@EnableHypermediaSupport` and registers a number of beans to ease building hypermedia-based applications, including a `LinkDiscoverers` (for client side support) and an `ObjectMapper` configured to correctly marshal responses into the desired representation. The `ObjectMapper` is customized by setting the various `spring.jackson.*` properties or, if one exists, by a `Jackson2ObjectMapperBuilder` bean.

You can take control of Spring HATEOAS's configuration by using `@EnableHypermediaSupport`. Note that doing so disables the `ObjectMapper` customization described earlier.

29.1.13 CORS Support

Cross-origin resource sharing (CORS) is a W3C specification implemented by most browsers that lets you specify in a flexible way what kind of crossdomain requests are authorized, instead of using some less secure and less powerful approaches such as IFRAME or JSONP.

As of version 4.2, Spring MVC supports CORS. Using controller method CORS configuration with `@CrossOrigin` annotations in your Spring Boot application does not require any specific configuration. Global CORS configuration can be defined by registering a `WebMvcConfigurer` bean with a customized `addCorsMappings(CorsRegistry)` method, as shown in the following example:

```
@Configuration
public class MyConfiguration {

	@Bean
	public WebMvcConfigurer corsConfigurer() {
		return new WebMvcConfigurer() {
			@Override
			public void addCorsMappings(CorsRegistry
registry) {
				registry.addMapping("/api/**");
			}
		};
	}
}
```

29.2 The “Spring WebFlux Framework”

Spring WebFlux is the new reactive web framework introduced in Spring Framework 5.0. Unlike Spring MVC, it does not require the Servlet API, is fully asynchronous and non-blocking, and implements the Reactive Streams specification through the Reactor project.

Spring WebFlux comes in two flavors: functional and annotation-based. The annotation-based one is quite close to the Spring MVC model, as shown in the following example:

```
@RestController
@RequestMapping("/users")
public class MyRestController {

	@GetMapping("/\{user}")
	public Mono<User> getUser(@PathVariable Long user) {
		// ...
	}

	@GetMapping("/\{user}/customers")
	public Flux<Customer> getUserCustomers(@PathVariable Long user)
{
		// ...
	}

	@DeleteMapping("/\{user}")
	public Mono<User> deleteUser(@PathVariable Long user) {
		// ...
	}

}
```

“WebFlux.fn”, the functional variant, separates the routing configuration from the actual handling of the requests, as shown in the following example:

```
@Configuration
public class RoutingConfiguration {

	@Bean
	public RouterFunction<ServerResponse>
monoRouterFunction(UserHandler userHandler) {
		return
route(GET("/\{user}").and(accept(APPLICATION_JSON)),
userHandler::getUser)

	.andRoute(GET("/\{user}/customers").and(accept(APPLICATION_JSON)
), userHandler::getUserCustomers)

	.andRoute(DELETE("/\{user}").and(accept(APPLICATION_JSON)),
userHandler::deleteUser);
	}

}

@Component
public class UserHandler {

	public Mono<ServerResponse> getUser(ServerRequest request) {
		// ...
	}

	public Mono<ServerResponse> getUserCustomers(ServerRequest
request) {
		// ...
	}

	public Mono<ServerResponse> deleteUser(ServerRequest request) {
		// ...
	}
}
```

WebFlux is part of the Spring Framework and detailed information is available in its reference documentation.

You can define as many `RouterFunction` beans as you like to modularize the definition of the router. Beans can be ordered if you need to apply a precedence.

To get started, add the `spring-boot-starter-webflux` module to your application.

Adding both `spring-boot-starter-web` and `spring-boot-starter-webflux` modules in your application results in Spring Boot auto-configuring Spring MVC, not WebFlux. This behavior has been chosen because many Spring developers add `spring-boot-starter-webflux` to their Spring MVC application to use the reactive `WebClient`. You can still enforce your choice by setting the chosen application type to `SpringApplication.setWebApplicationType(WebApplicationType.REACTIVE)`.

29.2.1 Spring WebFlux Auto-configuration

Spring Boot provides auto-configuration for Spring WebFlux that works well with most applications.

The auto-configuration adds the following features on top of Spring's defaults:

- Configuring codecs for `HttpMessageReader` and `HttpMessageWriter` instances (described later in this document).
- Support for serving static resources, including support for WebJars (described later in this document).

If you want to keep Spring Boot WebFlux features and you want to add additional WebFlux configuration, you can add your own `@Configuration` class of type `WebFluxConfigurer` but **without** `@EnableWebFlux`.

If you want to take complete control of Spring WebFlux, you can add your own `@Configuration` annotated with `@EnableWebFlux`.

29.2.2 HTTP Codecs with HttpMessageReaders and HttpMessageWriters

Spring WebFlux uses the `HttpMessageReader` and `HttpMessageWriter` interfaces to convert HTTP requests and responses. They are configured with `CodecConfigurer` to have sensible defaults by looking at the libraries available in your classpath.

Spring Boot applies further customization by using `CodecCustomizer` instances. For example, `spring.jackson.*` configuration keys are applied to the Jackson codec.

If you need to add or customize codecs, you can create a custom `CodecCustomizer` component, as shown in the following example:

```
import org.springframework.boot.web.codec.CodecCustomizer;

@Configuration
public class MyConfiguration {

	@Bean
	public CodecCustomizer myCodecCustomizer() {
		return codecConfigurer -> {
			// ...
		}
	}

}
```

You can also leverage Boot's custom JSON serializers and deserializers.

29.2.3 Static Content

By default, Spring Boot serves static content from a directory called `/static` (or `/public` or `/resources` or `/META-INF/resources`) in the classpath. It uses the `ResourceWebHandler` from Spring WebFlux so that you can modify that behavior by adding your own `WebFluxConfigurer` and overriding the `addResourceHandlers` method.

By default, resources are mapped on `/**`, but you can tune that by setting the `spring.webflux.static-path-pattern` property. For instance, relocating all resources to `/resources/**` can be achieved as follows:

```
spring.webflux.static-path-pattern=/resources/**
```

You can also customize the static resource locations by using `spring.resources.static-locations`. Doing so replaces the default values with a list of directory locations. If you do so, the default welcome page detection switches to your custom locations. So, if there is an `index.html` in any of your locations on startup, it is the home page of the application.

In addition to the “standard” static resource locations listed earlier, a special case is made for Webjars content. Any resources with a path in `/webjars/**` are served from jar files if they are packaged in the Webjars format.

Spring WebFlux applications do not strictly depend on the Servlet API, so they cannot be deployed as war files and do not use the `src/main/webapp` directory.

29.2.4 Template Engines

As well as REST web services, you can also use Spring WebFlux to serve dynamic HTML content. Spring WebFlux supports a variety of templating technologies, including Thymeleaf, FreeMarker, and Mustache.

Spring Boot includes auto-configuration support for the following templating engines:

- FreeMarker
- Thymeleaf
- Mustache

When you use one of these templating engines with the default configuration, your templates are picked up automatically from `src/main/resources/templates`.

29.2.5 Error Handling

Spring Boot provides a `WebExceptionHandler` that handles all errors in a sensible way. Its position in the processing order is immediately before the handlers provided by WebFlux, which are considered last. For machine clients, it produces a JSON response with details of the error, the HTTP status, and the exception message. For browser clients, there is a “whitelabel” error handler that renders the same data in HTML format. You can also provide your own HTML templates to display errors (see the next section).

The first step to customizing this feature often involves using the existing mechanism but replacing or augmenting the error contents. For that, you can add a bean of type `ErrorAttributes`.

To change the error handling behavior, you can implement `ErrorWebExceptionHandler` and register a bean definition of that type. Because a `WebExceptionHandler` is quite low-level, Spring Boot also provides a convenient `AbstractErrorWebExceptionHandler` to let you handle errors in a WebFlux functional way, as shown in the following example:

```
public class CustomErrorWebExceptionHandler extends
AbstractErrorWebExceptionHandler {

        // Define constructor here

        @Override
        protected RouterFunction<ServerResponse>
getRoutingFunction(ErrorAttributes errorAttributes) {

                return RouterFunctions
                                .route(aPredicate, aHandler)
                                .andRoute(anotherPredicate,
anotherHandler);
        }

}
```

For a more complete picture, you can also subclass `DefaultErrorWebExceptionHandler` directly and override specific methods.

Custom Error Pages

If you want to display a custom HTML error page for a given status code, you can add a file to an `/error` folder. Error pages can either be static HTML (that is, added under any of the static resource folders) or built with templates. The name of the file should be the exact status code or a series mask.

For example, to map `404` to a static HTML file, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- public/
             +- error/
             |   +- 404.html
             +- <other public assets>
```

To map all `5xx` errors by using a Mustache template, your folder structure would be as follows:

```
src/
 +- main/
     +- java/
     |   + <source code>
     +- resources/
         +- templates/
             +- error/
             |   +- 5xx.mustache
             +- <other templates>
```

29.2.6 Web Filters

Spring WebFlux provides a `WebFilter` interface that can be implemented to filter HTTP request-response exchanges. `WebFilter` beans found in the application context will be automatically used to filter each exchange.

Where the order of the filters is important they can implement `Ordered` or be annotated with `@Order`. Spring Boot auto-configuration may configure web filters for you. When it does so, the orders shown in the following table will be used:

Web Filter	Order
`MetricsWebFilter`	`Ordere`
`WebFilterChainProxy` (Spring Security)	`-100`
`HttpTraceWebFilter`	`Ordere`

29.3 JAX-RS and Jersey

If you prefer the JAX-RS programming model for REST endpoints, you can use one of the available implementations instead of Spring MVC. Jersey and Apache CXF work quite well out of the box. CXF requires you to register its `Servlet` or `Filter` as a `@Bean` in your application context. Jersey has some native Spring support, so we also provide auto-configuration support for it in Spring Boot, together with a starter.

To get started with Jersey, include the `spring-boot-starter-jersey` as a dependency and then you need one `@Bean` of type `ResourceConfig` in which you register all the endpoints, as shown in the following example:

```
@Component
```

```
public class JerseyConfig extends ResourceConfig {

	public JerseyConfig() {
		register(Endpoint.class);
	}

}
```

Jersey’s support for scanning executable archives is rather limited. For example, it cannot scan for endpoints in a package found in a fully executable jar file or in `WEB-INF/classes` when running an executable war file. To avoid this limitation, the `packages` method should not be used, and endpoints should be registered individually by using the `register` method, as shown in the preceding example.

For more advanced customizations, you can also register an arbitrary number of beans that implement `ResourceConfigCustomizer`.

All the registered endpoints should be `@Components` with HTTP resource annotations (`@GET` and others), as shown in the following example:

```
@Component
@Path("/hello")
public class Endpoint {

	@GET
	public String message() {
		return "Hello";
	}

}
```

Since the `Endpoint` is a Spring `@Component`, its lifecycle is managed by Spring and you can use the `@Autowired` annotation to inject dependencies and use the `@Value` annotation to inject external configuration. By default, the Jersey servlet is registered and mapped to `/*`. You can change the mapping by adding `@ApplicationPath` to your `ResourceConfig`.

By default, Jersey is set up as a Servlet in a `@Bean` of type `ServletRegistrationBean` named `jerseyServletRegistration`. By default, the servlet is initialized lazily, but you can customize that behavior by setting `spring.jersey.servlet.load-on-startup`. You can disable or override that bean by creating one of your own with the same name. You can also use a filter instead of a servlet by setting `spring.jersey.type=filter` (in which case, the `@Bean` to replace or override is `jerseyFilterRegistration`). The filter has an `@Order`, which you can set with `spring.jersey.filter.order`. Both the servlet and the filter registrations can be given init parameters by using `spring.jersey.init.*` to specify a map of properties.

There is a Jersey sample so that you can see how to set things up.

29.4 Embedded Servlet Container Support

Spring Boot includes support for embedded Tomcat, Jetty, and Undertow servers. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port `8080`.

29.4.1 Servlets, Filters, and listeners

When using an embedded servlet container, you can register servlets, filters, and all the listeners (such as `HttpSessionListener`) from the Servlet spec, either by using Spring beans or by scanning for Servlet components.

Registering Servlets, Filters, and Listeners as Spring Beans

Any `Servlet`, `Filter`, or servlet `*Listener` instance that is a Spring bean is registered with the embedded container. This can be particularly convenient if you want to refer to a value from your `application.properties` during configuration.

By default, if the context contains only a single Servlet, it is mapped to `/`. In the case of multiple servlet beans, the bean name is used as a path prefix. Filters map to `/*`.

If convention-based mapping is not flexible enough, you can use the `ServletRegistrationBean`, `FilterRegistrationBean`, and `ServletListenerRegistrationBean` classes for complete control.

Spring Boot ships with many auto-configurations that may define Filter beans. Here are a few examples of Filters and their respective order (lower order value means higher precedence):

Servlet Filter	Order
`OrderedCharacterEncodingFilter`	`Ordered.HIG`
`WebMvcMetricsFilter`	`Ordered.HIG`

Servlet Filter	Order
`ErrorPageFilter`	`Ordered.HIG`
`HttpTraceFilter`	`Ordered.LOW`

It is usually safe to leave Filter beans unordered.

If a specific order is required, you should avoid configuring a Filter that reads the request body at `Ordered.HIGHEST_PRECEDENCE`, since it might go against the character encoding configuration of your application. If a Servlet filter wraps the request, it should be configured with an order that is less than or equal to `OrderedFilter.REQUEST_WRAPPER_FILTER_MAX_ORDER`.

29.4.2 Servlet Context Initialization

Embedded servlet containers do not directly execute the Servlet 3.0+ `javax.servlet.ServletContainerInitializer` interface or Spring's `org.springframework.web.WebApplicationInitializer` interface. This is an intentional design decision intended to reduce the risk that third party libraries designed to run inside a war may break Spring Boot applications.

If you need to perform servlet context initialization in a Spring Boot application, you should register a bean that implements the `org.springframework.boot.web.servlet.ServletContextInitializer` interface. The single `onStartup` method provides access to the `ServletContext` and, if necessary, can easily be used as an adapter to an existing `WebApplicationInitializer`.

Scanning for Servlets, Filters, and listeners

When using an embedded container, automatic registration of classes annotated with `@WebServlet`, `@WebFilter`, and `@WebListener` can be enabled by using `@ServletComponentScan`.

> `@ServletComponentScan` has no effect in a standalone container, where the container's built-in discovery mechanisms are used instead.

29.4.3 The ServletWebServerApplicationContext

Under the hood, Spring Boot uses a different type of `ApplicationContext` for embedded servlet container support.

The `ServletWebServerApplicationContext` is a special type of `WebApplicationContext` that bootstraps itself by searching for a single `ServletWebServerFactory` bean. Usually a `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` has been auto-configured.

You usually do not need to be aware of these implementation classes. Most applications are auto-configured, and the appropriate `ApplicationContext` and `ServletWebServerFactory` are created on your behalf.

29.4.4 Customizing Embedded Servlet Containers

Common servlet container settings can be configured by using Spring `Environment` properties. Usually, you would define the properties in your `application.properties` file.

Common server settings include:

- Network settings: Listen port for incoming HTTP requests (`server.port`), interface address to bind to `server.address`, and so on.
- Session settings: Whether the session is persistent (`server.servlet.session.persistent`), session timeout (`server.servlet.session.timeout`), location of session data (`server.servlet.session.store-dir`), and session-cookie configuration (`server.servlet.session.cookie.*`).
- Error management: Location of the error page (`server.error.path`) and so on.
- SSL
- HTTP compression

Spring Boot tries as much as possible to expose common settings, but this is not always possible. For those cases, dedicated namespaces offer server-specific customizations (see `server.tomcat` and `server.undertow`). For instance, access logs can be configured with specific features of the embedded servlet container.

See the `ServerProperties` class for a complete list.

Programmatic Customization

If you need to programmatically configure your embedded servlet container, you can register a Spring bean that implements the `WebServerFactoryCustomizer` interface. `WebServerFactoryCustomizer` provides access to the `ConfigurableServletWebServerFactory`, which includes numerous customization setter methods. The following example shows programmatically setting the port:

```
import org.springframework.boot.web.server.WebServerFactoryCustomizer;
import
org.springframework.boot.web.servlet.server.ConfigurableServletWebServe
rFactory;
import org.springframework.stereotype.Component;

@Component
public class CustomizationBean implements
WebServerFactoryCustomizer<ConfigurableServletWebServerFactory> {

	@Override
	public void customize(ConfigurableServletWebServerFactory
server) {
		server.setPort(9000);
	}

}
```

`TomcatServletWebServerFactory`, `JettyServletWebServerFactory` and `UndertowServletWebServerFactory` are dedicated variants of `ConfigurableServletWebServerFactory` that have additional customization setter methods for Tomcat, Jetty and Undertow respectively.

Customizing ConfigurableServletWebServerFactory Directly

If the preceding customization techniques are too limited, you can register the `TomcatServletWebServerFactory`, `JettyServletWebServerFactory`, or `UndertowServletWebServerFactory` bean yourself.

```
@Bean
public ConfigurableServletWebServerFactory webServerFactory() {
	TomcatServletWebServerFactory factory = new
TomcatServletWebServerFactory();
	factory.setPort(9000);
	factory.setSessionTimeout(10, TimeUnit.MINUTES);
```

```
        factory.addErrorPages(new ErrorPage(HttpStatus.NOT_FOUND,
"/notfound.html"));
        return factory;
}
```

Setters are provided for many configuration options. Several protected method “hooks” are also provided should you need to do something more exotic. See the source code documentation for details.

29.4.5 JSP Limitations

When running a Spring Boot application that uses an embedded servlet container (and is packaged as an executable archive), there are some limitations in the JSP support.

- With Jetty and Tomcat, it should work if you use war packaging. An executable war will work when launched with `java -jar`, and will also be deployable to any standard container. JSPs are not supported when using an executable jar.
- Undertow does not support JSPs.
- Creating a custom `error.jsp` page does not override the default view for error handling. Custom error pages should be used instead.

There is a JSP sample so that you can see how to set things up.

29.5 Embedded Reactive Server Support

Spring Boot includes support for the following embedded reactive web servers: Reactor Netty, Tomcat, Jetty, and Undertow. Most developers use the appropriate “Starter” to obtain a fully configured instance. By default, the embedded server listens for HTTP requests on port 8080.

29.6 Reactive Server Resources Configuration

When auto-configuring a Reactor Netty or Jetty server, Spring Boot will create specific beans that will provide HTTP resources to the server instance: `ReactorResourceFactory` or `JettyResourceFactory`.

By default, those resources will be also shared with the Reactor Netty and Jetty clients for optimal performances, given:

- the same technology is used for server and client
- the client instance is built using the `WebClient.Builder` bean auto-configured by Spring Boot

Developers can override the resource configuration for Jetty and Reactor Netty by providing a custom `ReactorResourceFactory` or `JettyResourceFactory` bean - this will be applied to both clients and servers.