**Spring Boot Important points**

1. By default, Tomcat is embedded as default server in spring-web-starter. We can use <exclusion/> tag to give chance to other servers. Accordingly, we can add starters for other servers.

e.g.,

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </exclusion>  </exclusions>  </dependency> |

1. To inform Spring that we are using some external server we should use **<scope>provided</scope>** tag with external server dependency.

e.g.,

|  |
| --- |
| <dependency>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-web</artifactId>  <exclusions>  <exclusion>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-starter-tomcat</artifactId>  </exclusion>  </exclusions>  </dependency> |

1. If we want to ensure that our WAR files are also executable, we need to include the spring-boot-maven-plugin. This plugin in Maven lets you package executable JAR or WAR archives to run an application.

e.g.,

|  |
| --- |
| <build>  <plugins>  <plugin>  <groupId>org.springframework.boot</groupId>  <artifactId>spring-boot-maven-plugin</artifactId>  </plugin>  </plugins>  </build> |

1. In order to create a deployable WAR file, we need to make a few changes to our application’s code as well. Our main method will now extend SpringBootServletInitializer to support war file deployment to external servers.
2. **SpringBootServletInitializer** is an opinionated **WebApplicationInitializer** to run a Spring Application from a traditional WAR deployment. It binds Servlet, Filter & ServletContextInitializer beans from the application context to the server.
3. To configure the application either override the method:

**configure (SpringApplicationBuilder** springApplicationBuilder**)**

& call **springApplicationBuilder.sources (**MainClassName**.class)**

or make the **initializer itself a @Configuration**.

1. If we are using **SpringBootServletInitializer** in combination with other **WebApplicationInitializers**, we might also want to add an **@Ordered** annotation to configure a specific startup order.

e.g.,

|  |
| --- |
| **import** org.springframework.boot.SpringApplication;  **import** org.springframework.boot.autoconfigure.SpringBootApplication;  **import** org.springframework.boot.builder.SpringApplicationBuilder;  **import** org.springframework.boot.web.servlet.support.SpringBootServletInitializer;  @SpringBootApplication  **public** **class** GenericMainClass **extends** SpringBootServletInitializer {  **protected** SpringApplicationBuilder configure(SpringApplicationBuilder application) {  **return** application.sources(GenericMainClass.**class**);  }  **public** **static** **void** main(String[] args) {  SpringApplication.*run*(GenericMainClass.**class**, args);  System.***out***.println("Hello Spring");  }  } |

1. Note that a **WebApplicationInitializer** is **only needed if we are building a war file & deploying it**. If we prefer to run an embedded web server then we won’t need this at all.
2. Thymeleaf is a Java-based template engine typically used in Spring Boot applications used to transform & render the data that you want to display in your web apps.
3. Thymeleaf is a server-side template engine & it provides great support for rendering HTML5.
4. Using the Thymeleaf starter template, automatically sets up a view resolver for our Spring boot web application.
5. It also allows us to use message properties configured in special files.
6. Maven archetype is a project template which contains the basic structure of our Maven project.
7. The Spring Data JPA starter **(spring-boot-starter-data-jpa)** template allows you to persist data in a SQL database using the Java Persistence API & the frameworks Spring Data & Hibernate.
8. Hibernate is an object relational mapping (ORM) framework, which maps your java objects or classes to underlying database tables.
9. JPA or the Java Persistence API is what we’ll use to work with Hibernate & Spring data does a lot of magic for us i.e., basically provides implementation for many common database operations for our table.
10. Now the Hibernate ORM framework connects to the underlying database table using JDBC & for that we need to add the MySQL driver as dependency. **(mysql-connector-j)**

**JPA Related Annotations**

1. @Entity

* This tags the class as mapped to an underlying database table. e.g., **@Entity(name=”Product”**) where table name is Product.
* For class annotated with @Entity(“Product”), every instance of this class maps to a record or a row in the Product table.
* The class member defined in the entity class corresponds to the columns of the database table.

1. @Id

* Every entity in JPA has to have a primary key. The class member variable annotated with @Id is specified as Primary key.

1. @GeneratedValue (strategy=GenerationType.IDENTITY)

* It means the JPA & Hibernate will automatically generate values for this primary key.

1. @Repository

* This basically tells Spring that this is a spring managed component that will be used to access data from an underlying database table.
* Exceptions thrown by this component will be converted to Data Access exceptions by Spring.

1. CrudRepository<T, ID> interface where T is table type, ID is primary key type

* It is a generic interface with 2 generic type parameters.
* It exposes create, read, update, delete operation for the specified table.
* E.g., **CrudRepository<Product, Long>** means the CrudRepository interface will expose create, read, update & delete operations for the Product entity & this Product entity has a primary key of type Long.

Q. If CrudRepository is an interface, where is the implementation?

Ans:

|  |
| --- |
| import org.springframework.data.repository.CrudRepository;  import org.springframework.stereotype.Repository;  @**Repository**  public interface **ProductRepository** extends **CrudRepository**<Product, Long> {  } |

* It turns out Spring Data takes care of all the implementation, we don’t have to specify implementation by ourselves. We simply need to extend the CrudRepository interface, specify the type of entity & type of primary key & Spring Data module will auto generate the implementation for us.
* The implementation for all of the methods exposed by the CrudRepository will be automatically available to us like findById(), findAll(), save(), saveAll(), delete(), deleteAll(), existsById() etc.

**Spring Boot Caching**

1. @EnableCaching

* Class level annotation
* This tells Spring that caching is enable for this app.
* We can decide which layer we want our caching to be performed.
* Thumb rule: Specify caching annotation in a layer that is common to all of your client.
* E.g., Let’s choose Service layer for caching so that not only our REST API can take advantage of caching, but also the web UI.

1. @Cacheable

* Method level annotation
* If response is present in the cache it is returned directly, the code is not executed.
* E.g.,

|  |
| --- |
| @**Cacheable**("products")  **public** List<Product> getAllProducts() {  List<Product> products = **new** ArrayList<>();  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  productRepository.findAll().forEach(products::add);  **return** products;  } |

* Here @Cacheable(“products”) means products will be the name of the cache where the response from getAllProducts () method will be stored.

**Cache working:**

* If the list of products i.e., the response of this method is present within the products cache, the response is returned directly to the user. The code within this method will not be executed.
* When we invoke this method for the first time or if the cache has been previously cleaned up (i.e., the response for this method will not be present in the cache), in this case the code for this method will be executed along with the sleep of 3 seconds, & the response will be cached in the products cache.

e.g.,

|  |
| --- |
| @**Cacheable**(**value**="product", **key**="#p0")  **public** Optional<Product> getProduct(Long id) {  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  **return** productRepository.findById(id);  } |

* Here, the name of the cache where we’ll store this product retrieve by ID is **product**. In addition, we also specify a key that’ll be used to store the response for one invocation of this method.
* The **key=”#p0”** means the key will be the first argument to this method, which is the ID of the product. The response of this method will be stored in the cache keyed by the ID of the product.

1. @CacheEvict(value=””, allEntries=true)

* Method level annotation.
* The problem with making all methods @Cacheable is size. We don’t want to populate the cache with values that we don’t need often. Cache can grow quite large, quite fast & we could be holding on to a lot of stale or unused data.
* We can use the @CacheEvict annotation to indicate the removal of one or more /all values so that fresh values can be loaded into the cache again:

e.g.,

|  |
| --- |
| @CacheEvict(value="products", allEntries=**true**)  **public** **void** addProduct(Product product) {  **try** {  Thread.*sleep*(3000);  } **catch** (InterruptedException e) {  e.printStackTrace();  }  productRepository.save(product);  } |

* Here, we perform a cache eviction on the cache named **products**. The cache “**products**” holds a list of all of the products that we have.
* Also note that we evict allEntries within the products cache set to true i.e., this cache will be completely cleared so that the new product will be visible in the response next time we retrieve all products.

1. @Caching(evict={@CacheEvict(value=”’, key=””), @CacheEvict(value=””, allEntries=true})

**Java Mail Service**

a) spring-boot-starter-mail dependency allows us to use the JavaMailSender service.

Note: For using gmail, enable IMAP from all setting in GMAIL.

b) IMAP is what our Java mail sending service uses in order to send emails.

c) Use app code for gmail id as password in configuration.

d) Some configs in **application.properties**

|  |
| --- |
| *spring.mail.host*=smtp.gmail.com  *spring.mail.port*=587  *spring.mail.username*=  *spring.mail.password*=  *spring.mail.properties.mail.smtp.starttls.enable*=true  *spring.mail.properties.mail.smtp.auth*=true  *spring.mail.properties.mail.smtp.connectiontimeout*=5000  *spring.mail.properties.mail.smtp.timeout*=5000  *spring.mail.properties.mail.smtp.writetimeout*=5000  *spring.mail.properties.mail.smtip.starttls.enable*=true |

e) e.g., Email service with & without attachment

|  |
| --- |
| **import** javax.mail.MessagingException;  **import** javax.mail.internet.MimeMessage;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.boot.CommandLineRunner;  **import** org.springframework.context.annotation.Configuration;  **import** org.springframework.core.io.ClassPathResource;  **import** org.springframework.mail.SimpleMailMessage;  **import** org.springframework.mail.javamail.JavaMailSender;  **import** org.springframework.mail.javamail.MimeMessageHelper;  @Configuration  **public** **class** MailController **implements** CommandLineRunner {  @Autowired  **private** JavaMailSender javaMailSender;    **public** **void** run(String... args) **throws** MessagingException, IOException {  System.***out***.println("Sending mail...");  sendEmail();  sendEmailWithAttachment();  System.***out***.println("Done");  }  **private** **void** sendEmailWithAttachment() **throws** MessagingException {  System.***out***.println("Mail with attachement sending....");  MimeMessage mimeMsg = javaMailSender.createMimeMessage();  MimeMessageHelper mimeMsgHelper = **new** MimeMessageHelper(mimeMsg, **true**);    mimeMsgHelper.setTo("mail2shivamraj@gmail.com");    mimeMsgHelper.setSubject("Here is a attached pic!");  mimeMsgHelper.setText("<h3>Take a look at the attachment:-)</h3>", **true**);    mimeMsgHelper.addAttachment("coder.jpg", **new** ClassPathResource("th.jpg"));    javaMailSender.send(mimeMsg);  System.***out***.println("Mail with attachment sent!");  }  **private** **void** sendEmail() {    SimpleMailMessage msg = **new** SimpleMailMessage();  msg.setTo("mail2shivamraj@gmail.com");    msg.setSubject("Email sent using SpringBoot");  msg.setText("Hello! \n\n Welcome to SpringBoot, easy to send email wasn't it?");    javaMailSender.send(msg);  }  } |

**Interceptor**

* Interceptors are an integral part of the Spring MVC web framework.
* Interceptors intercept client requests to handler mappings & process them
* When working with Spring MVC, any incoming web request is sent to the dispatcher server, which then looks up the handler mappings that we have configured to find the right method to handle the particular incoming request. There is where interceptors come in.
* Interceptors in Spring allow us to intercept client requests to handler mapping & process these requests before they are handled by our controller methods..
* We can use interceptors to make sure all of the input parameters are in the right format or add additional parameters that we want to send to our handler.
* We can also use interceptors to perform some kind of action in the background for every client request.

Working of Spring boot interceptor

* We need to implement HandlerInterceptor interface in order to override its preHandle, postHandle & afterCompletion methods.

1. preHandle ()

* This method is executed **before a request is sent to a handler**.
* We can inject bits of information from other components in our system that our handler might need.

1. postHandle ()

* This method is executed **after the request is sent to the handler.**
* The postHandle() method is invoked after our handler code has been executed but before the view is rendered to the user.
* This handler is a great place to add-in any additional model parameters that we want the view to understand & display.

1. afterCompletion()

* This method is executed **after the completion of the request when the view is rendered.**
* This method is invoked once the response is completely processed by our handler. So, after the handler code is executed, when the request is complete & the view has been rendered, that’s when afterCompletion() method is called.
* Within this handler, we have not just the request & response but also information about any exceptions that were thrown.
* We can use this for logging or for timing purposes.

**Note**: Before these interceptors becomes part of our spring application, it has to be explicitly resgistered as such & we can do this by implementing **WebMvcConfigurer** interface & overriding addInterceptors() method.

|  |
| --- |
| @Component  **public** **class** WebAppConfiguration **implements** WebMvcConfigurer {  @Autowired  **private** BookHandlerInterceptor bookInterceptor;    @Override  **public** **void** addInterceptors(InterceptorRegistry registry) {  registry.addInterceptor(bookInterceptor);  }  } |

e.g.,

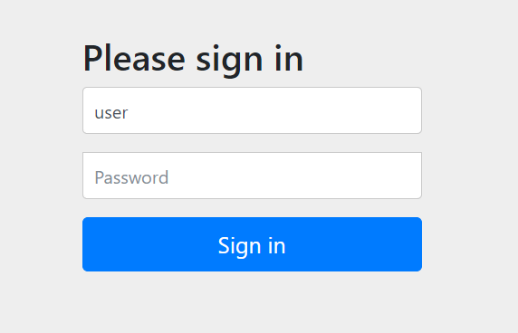
|  |
| --- |
| **import** javax.servlet.http.HttpServletRequest;  **import** javax.servlet.http.HttpServletResponse;  **import** org.springframework.beans.factory.annotation.Autowired;  **import** org.springframework.mail.SimpleMailMessage;  **import** org.springframework.mail.javamail.JavaMailSender;  **import** org.springframework.stereotype.Component;  **import** org.springframework.web.servlet.HandlerInterceptor;  **import** org.springframework.web.servlet.ModelAndView;  @Component  **public** **class** BookHandlerInterceptor **implements** HandlerInterceptor {  @Autowired  **private** JavaMailSender javaMailSender;  **private** **static** **final** SimpleDateFormat ***dateFormat*** = **new** SimpleDateFormat("MM/dd/yyyy HH:mm:ss");  @Override  **public** **boolean** preHandle(HttpServletRequest request, HttpServletResponse response, Object handler)  **throws** Exception {  **if** (request.getParameter("bookId") != **null**) {  System.***out***.println("preHandle() method sending book access mail...");  sendEmail(request.getParameter("bookId"), "Book accessed");  System.***out***.println("Done");  }  **return** **true**;  }  @Override  **public** **void** postHandle(HttpServletRequest request, HttpServletResponse response, Object handler,  ModelAndView modelAndView) **throws** Exception {  **if** (request.getParameter("bookId") != **null**) {  System.***out***.println("postHandle() method sending book access mail...");  sendEmail(request.getParameter("bookId"), "Book access complete");  System.***out***.println("Done");  }  }    @Override  **public** **void** afterCompletion(HttpServletRequest request, HttpServletResponse response, Object handler,  Exception ex) **throws** Exception {  **if** (request.getParameter("bookId") != **null**) {  System.***out***.println("Sending book request & response completion mail...");  sendEmail(request.getParameter("bookId"), "Request & Response complete");  System.***out***.println("Done");  }  }  **private** **void** sendEmail(String bookId, String messageText) {  SimpleMailMessage msg = **new** SimpleMailMessage();  msg.setTo("mail2shivamraj@gmail.com");  msg.setSubject("Book related activity for book: " + bookId);  msg.setText(messageText + ": " + ***dateFormat***.format(**new** Date()));    javaMailSender.send(msg);    }  } |

**API Proxies for services**

* An API proxy is service that sits in front of our APIs i.e., rather than have our users consume our APIs directly, we’ll have them send their request to this API proxy.
* API proxies control access to our APIs for security, monetization by tracking how many times API requests are made by specific users & general access tracking.
* So, using API proxies for our APIs is quite common in the real world.
* We can do API proxy in Spring boot by using Zuul’s routing & filtering.

**Spring Boot Starter Security**

* We need **spring-boot-starter-security** dependency for security-based functions in our project.
* The way Spring security works is that if spring security is present on the classpath, Spring Boot automatically secures all HTTP endpoints with basic authentication.
* With Spring Security, all of our web pages will be behind a login screen automatically where default username is user (by default spring security autoconfigures a user for us behind the scene) & pass is present as key on the console. This is Spring security default login page.



* Spring security sets up a filter security interceptor which intercepts all the requests that we make to this application & ensures that we’re logged in before the request is satisfied.
* We can configure new user by adding configuration in application.properties file

|  |
| --- |
| spring.security.user.name=user  spring.security.user.password=springboot |

* With Spring security, our application can have multiple users belonging to different roles such as regular user, an administrator or so on..
* We can also configure certain pages as begin behind a login & other pages are freely accessible.
* As a best practice, we need store passwords in plain text either within application or database. We always encrypt the password. We can use one of the password encoders that Spring security offers like BCryptPasswordEncoder (it uses BCrypt hashing algorithm to encode passwords)
* In order to configure security setting for our web application, we need to create some beans for our SecurityConfig & use @Configuration on the SecurityConfig class.

e.g.,

|  |
| --- |
| @**Configuration**  **public** **class** SecurityConfig {  @**Bean**  **public** PasswordEncoder passwordEncoder() {  **return** **new** BCryptPasswordEncoder();  }  @**Bean**  **public** UserDetailsService userDetailsService() {  UserDetails user = User.*withUsername*("loonyuser")  .password("$2a$10$nBIb9HWMo64Ljrncfb").roles("USER").build();  UserDetails admin = User.*withUsername*("loonyadmin")  .password("$2a$10$yWQYUaAVUg ").roles("ADMIN").build();  **return** **new** InMemoryUserDetailsManager(user, admin);  }    @**Bean**  **public** SecurityFilterChain filterChain(HttpSecurity http) **throws** Exception {  http.authorizeRequests().antMatchers("/").permitAll()  .antMatchers("/admin").hasRole("ADMIN")  .anyRequest().authenticated()  .and()  .formLogin()  .loginPage("/login")  .permitAll()  .and()  .logout().permitAll();  **return** http.build();  }  } |

* Spring framework encourages users to move towards a component-based security configuration. So, Instead of extending **WebSecurityConfigurerAdapter** & overriding methods for configuring HttpSecurity & WebSecurity as in the old way – Now we have to declare 2 beans of type **SecurityFilterChain** & **WebSecurityCustomizer**.
* Refer: https://codejava.net/frameworks/spring-boot/fix-websecurityconfigureradapter-deprecated