Spring MVC

- The Spring Web model-view-controller (MVC) framework is designed around a DispatcherServlet that dispatches requests to handlers and then to view resolver for view resolution.
- The default handler is based on the @Controller and @RequestMapping annotations, offering a wide range of flexible handling methods.
- Annotation @PathVariable provides support for restful web services.

DispatcherServlet

- ❑ DispatcherServlet is a servlet configured in web.xml
- ❑ It uses special beans to process request and render views
- ❑ It supports the following init-param elements for configuration
- ❑ By default it look for a file with name {servlet-name}-servlet.xml

contextClass
Class that implements WebApplicationContext, which instantiates the context used by this Servlet. By default, the XmlWebApplicationContext is used.

contextConfigLocation
String that is passed to the context instance to indicate where context(s) can be found. The string consists potentially of multiple strings (using a comma as a delimiter) to support multiple contexts. Incase of multiple context locations with beans that are defined twice, the latest location takes precedence.

namespace
Namespace of the WebApplicationContext. Defaults to [servlet-name]-servlet.

Special Beans

HandlerMapping

Maps incoming requests to handlers and a list of pre- and post-processors (handler interceptors) based on some criteria the details of which vary by HandlerMapping implementation.

ViewResolver

Resolves logical String-based view names to actual View types.

LocaleResolver

Resolves the locale a client is using and possibly their time zone, in order to be able to offer internationalized views

ThemeResolver

Resolves themes your web application can use

MultipartResolver

Parses multi-part requests for example to support processing file uploads from HTML forms.

FlashMapManager

Stores and retrieves the "input" and the "output” FlashMap that can be used to pass attributes from one request to another, usually across a redirect.

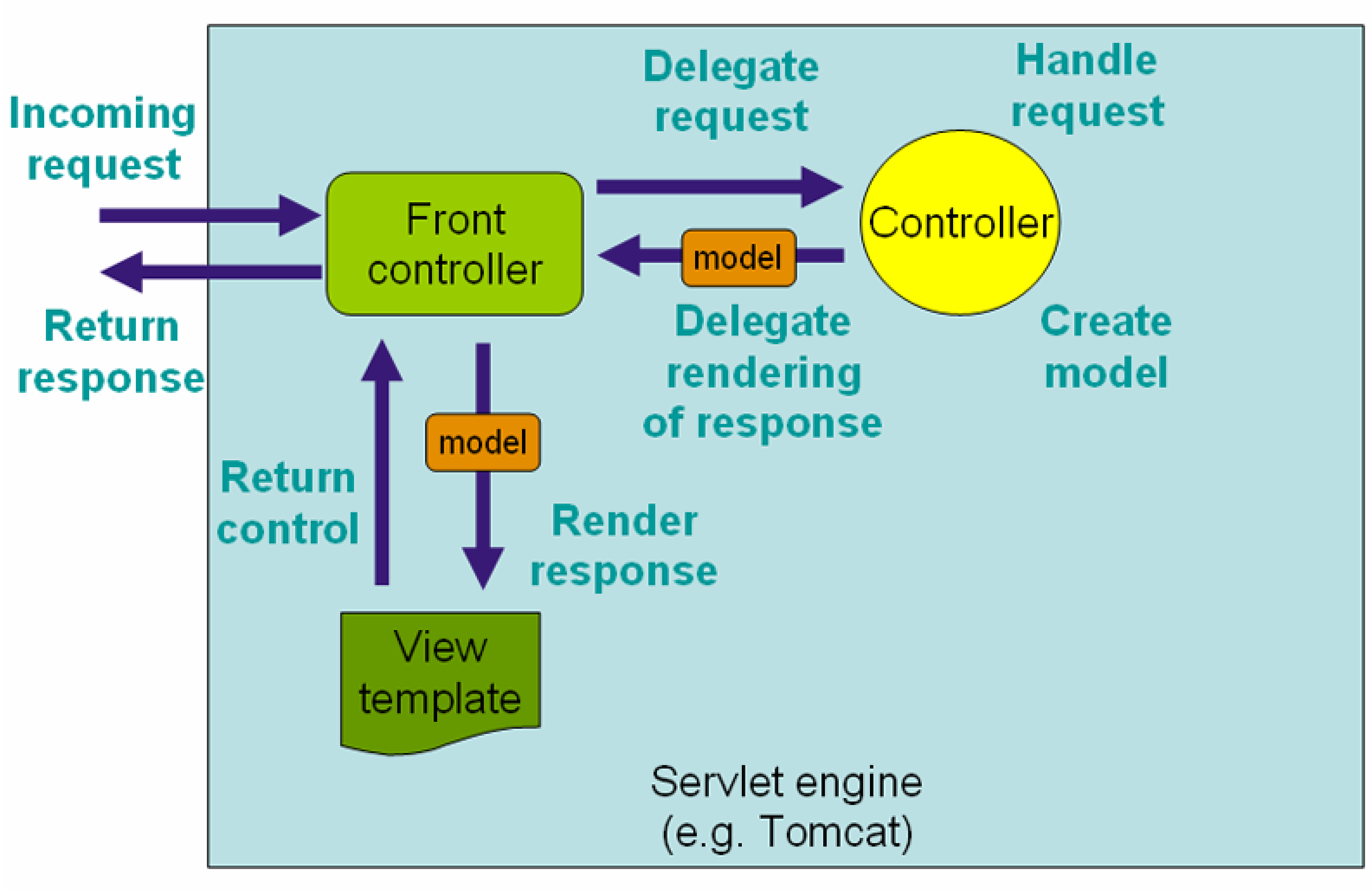
DispatcherServlet Process

- The WebApplicationContext is searched for and bound in the request as an attribute that
- the controller and other elements in the process can use. It is bound by default under the key DispatcherServlet.WEB_APPLICATION_CONTEXT_ATTRIBUTE.
- The locale resolver is bound to the request to enable elements in the process to resolve the locale to use when processing the request
- The theme resolver is bound to the request to let elements such as views determine which theme to use.
- If you specify a multipart file resolver, the request is inspected for multiparts; if multiparts are found, the request is wrapped in a MultipartHttpServletRequest for further processing by other elements in the process.
- An appropriate handler is searched for. If a handler is found, the execution chain associated with the handler (preprocessors, postprocessors, and controllers) is executed in order to prepare a model or rendering.
- If a model is returned, the view is rendered. If no model is returned, no view is rendered, because the request could already have been fulfilled.

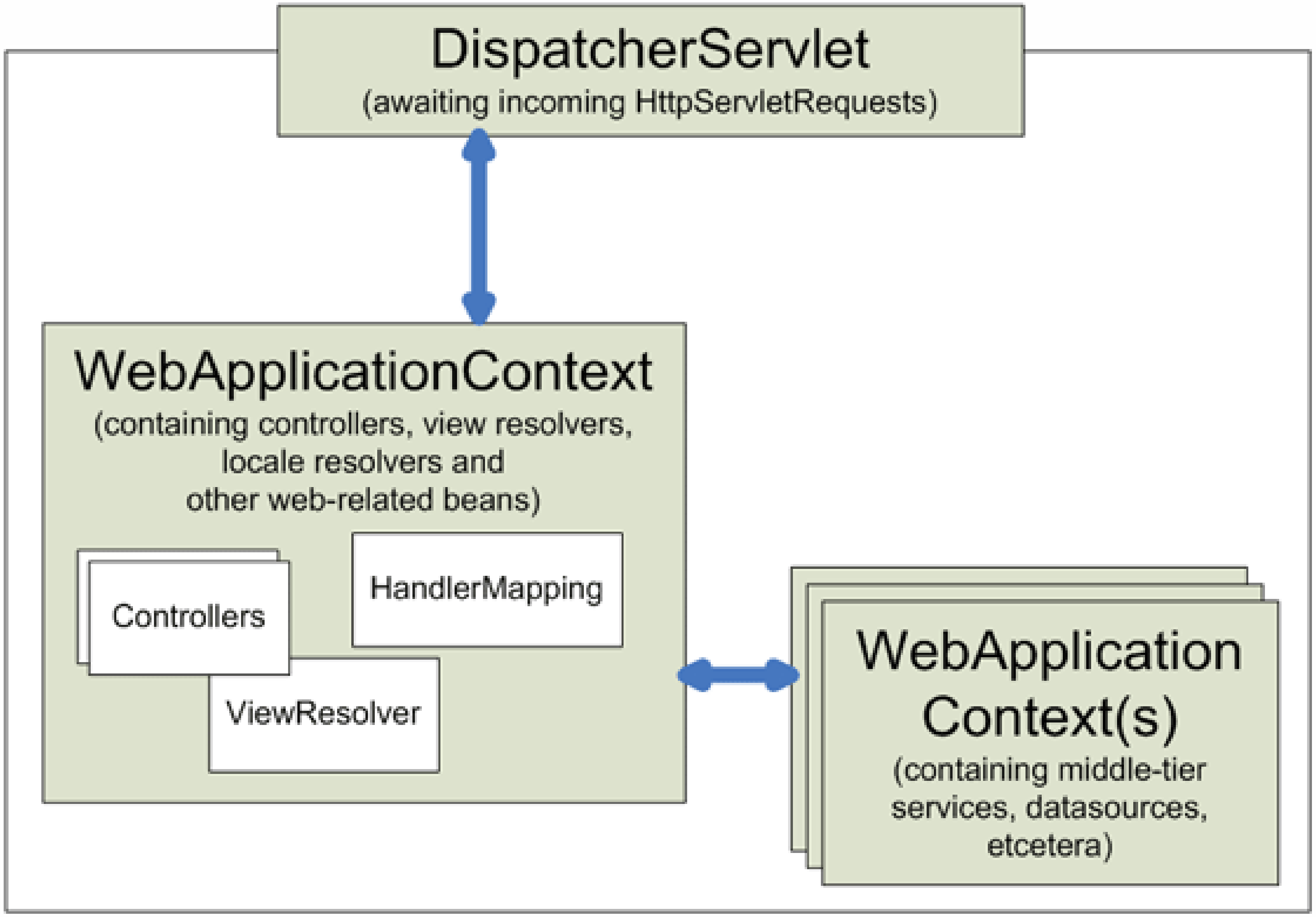
Spring MVC Components

- ❑ A Controller is typically responsible for preparing a model Map with data and selecting a view name.
- ❑ View name resolution is handled by View Resolver.
- ❑ The model is a Map interface, which allows for the complete abstraction of the view technology.
- ❑ You can integrate directly with template based rendering technologies such as JSP, Velocity and Freemarker.
- ❑ The model Map is simply transformed into an appropriate format, such as JSP request attributes.
- ❑ It is also possible to generate XML or JSON and send it to client.

Request Processing Workflow



Context Hierarchy


DispatcherServlet
(awaiting incoming HttpServletRequests)
WebApplicationContext
(containing controllers, view resolvers, locale resolvers and other web-related beans)
Controllers
HandlerMapping
ViewResolver
WebApplication Context(s)
(containing middle-tier services, datasources, etcetera)

Controller

- Controllers provide access to the application behavior
- Controllers interpret user input and transform it into a model that is represented to the user by the view.
- A controller is a Spring bean defined using @Controller annotation.
- It has methods that are mapped to requests using @RequestMapping annotations.
- We have to place controllers in packages that are specified using <component-scan> element in configuration file.
- Controllers need not extend any class or implement any interface.
- Methods in controller can have flexible signatures .

@RequestMapping

- You use the @RequestMapping annotation to map URLs such as /login onto an entire class or a particular handler method.
- Typically the class-level annotation maps a specific request path (or path pattern)
- Additional method-level annotations narrowing the primary mapping for a specific HTTP method request method ("GET", "POST", etc.) or an HTTP request parameter condition
- It has two properties - value and method.
- Method can be set to one of the options in RequestMethod enumeration representing HTTP request methods

Controller Example

```
@Controller
@RequestMapping("/books")
public class BooksController {
  @Autowired
  public BooksController(Catalog cat) {
  }
  @RequestMapping(method = RequestMethod.GET)
  public String  get() {
      // process
  }
  @RequestMapping(value="/new", method = RequestMethod.GET)
  public Book  newBook() {
      // process
  }
  @RequestMapping(method = RequestMethod.POST)
  public String addBook(@Valid Book book,BindingResult result){
     // process
  }
}
```

@RequestMapping Handler methods

An @RequestMapping handler method can have a very flexible signatures. Most arguments can be used in arbitrary order with the only exception of BindingResult arguments.

@RequestMapping Handler method Parameters

- Request or response objects (Servlet API).
- Session object (Servlet API): of type HttpSession.
- java.io.InputStream / java.io.Reader for access to the request’s content.
- java.io.OutputStream / java.io.Writer for generating the response’s content.
- @PathVariable annotated parameters for access to URI template variables.
- @RequestParam annotated parameters for access to specific Servlet request parameters
- @RequestHeader annotated parameters for access to specific Servlet request HTTP headers.
- @RequestBody annotated parameters for access to the HTTP request body.
- @RequestPart annotated parameters for access to the content of a "multipart/form-data" request part.
- java.util.Map / org.springframework.ui.Model /org.springframework.ui.ModelMap for enriching the implicit model that is exposed to the web view.

@RequestMapping Handler method return types

- A ModelAndView object
- A Model object
- A Map object for exposing a model
- A View object
- A String value that is interpreted as the logical view name
- void if the method handles the response itself (by writing the response content directly, declaring an argument of type ServletResponse / HttpServletResponse for that purpose) or if the view name is supposed to be implicitly determined through a RequestToViewNameTranslator.
- Any other return type is considered to be a single model attribute to be exposed to the view, using the attribute name specified through @ModelAttribute at the method level (or the default attribute name based on the return type class name). The model is implicitly enriched with command objects and the results of @ModelAttribute annotated reference data accessor methods

@ModelAttribute

- @ModelAttribute can be used on methods or on method arguments.
- It indicates the argument should be retrieved from the model. If not present in the model, the argument should be instantiated first and then added to the model.
- Once present in the model, the argument's fields should be populated from all request parameters that have matching names.
- This is known as data binding in Spring MVC

```
@RequestMapping ( value="/jobs/edit",
                  method = RequestMethod.POST)
public String processSubmit(@ModelAttribute Job job)
{
}
```

@RequestParam

- Annotation which indicates that a method parameter should be bound to a web request parameter.
- If the method parameter is Map<String, String> or MultiValueMap<String, String> and a parameter name is not specified, then the map parameter is populated with all request parameter names and values.
- Optional attributes - defaultValue, name, required
- When defaultValue is provided, it implicitly sets required to false.

```
@RequestMapping(value="/books", method=RequestMethod.GET)
public String findBook(@RequestParam("id") String bookId)
{
}
@RequestMapping(value="/chapter”, method=RequestMethod.GET)
public String findChapter(
 @RequestParam(name=“id”, defaultValue=“1”) int id,
 @RequestParam(name=“chno”) int chno) {

}
```

@PathVariable

- ❑ **A URI Template is a URI-like string, containing one or more variable names Variables are enclosed in {}**
- ❑ **We can use @PathVariable annotation on a method argument to bind it to the value of a URI template variable**

```
@RequestMapping(value="/books/{id}",
                method=RequestMethod.GET)
public String findBook(@PathVariable("id")  String bookId,
                       Model model) {

}

@RequestMapping(value="/books/{id}/chapters/{chno}",
                method=RequestMethod.GET)
public String findChapter(@PathVariable String id,
                          @PathVariable String chno,
                          Model model) {

}
```

Form Handling

- Parameter for request handler could be a class.
- All request parameters with same names as properties are automatically copied.

```
@RequestMapping(value="/add", method = RequestMethod.POST)
public String addBook(Book book, Model model) {

}
```

Validation of Form Fields

- ❑ Spring supports Validation API
- ❑ Just make sure you have implementation of validation API, such as Hibernate Validator, in classpath.
- ❑ Annotate fields in model class with validation annotations.
- ❑ Use @Valid annotation for class in request handler.
- ❑ Provide Errors or BindingResult object to catch errors.

```
@RequestMapping(value="/add", method = RequestMethod.POST)
public String addBook(@Valid Book book, Errors errors) {

}
```

Other Annotations related to Request Handler

- @SessionAttributes
- @RequestParam
- @CookieValue
- @RequestHeader
- @ModelAttribute
- @RequestPart
- @RequestBody

Available View Resolvers

ResourceBundleViewResolver

Uses bean definitions in a ResourceBundle, specified by the bundle base name. Typically you define the bundle in a properties file, located in the classpath. The default file name is views.properties.

UrlBasedViewResolver

Effects the direct resolution of logical view names to URLs, without an explicit mapping definition. This is appropriate if your logical names match the names of your view resources in a straightforward manner, without the need for arbitrary mappings.

InternalResourceViewResolver

Convenient subclass of UrlBasedViewResolver that supports InternalResourceView (in effect, Servlets and JSPs) and subclasses such as JstlView and TilesView. You can specify the view class for all views generated by this resolver by using setViewClass(..)

Prefixes

redirect : prefix

If a view name is returned that has the prefix redirect:, the UrlBasedViewResolver (and all subclasses) will recognize this as a special indication that a redirect is needed. The rest of the view name will be treated as the redirect URL.

forward: prefix

It is also possible to use a special forward: prefix for view names that are ultimately resolved by UrlBasedViewResolver and subclasses. This creates an InternalResourceView (which ultimately does a RequestDispatcher.forward()) around the rest of the view name, which is considered a URL.

```
@RequestMapping(value = "/files",
                method = RequestMethod.POST)
public String upload(...) {
     // ...
     return "redirect:files}";
}
```

Spring Form Tags

- Spring provides comprehensive set of data binding-aware tags for handling form elements
- Form tags are bundled with spring-webmvc.jar
- Register form tags as follows :

```
<%@ taglib prefix=”sf" uri="http://www.springframework.org/tags/form" %>
```

Spring Form Tags

JSP tag	Description
`<sf:checkbox>`	Renders an HTML `<input>` tag with `type` set to `checkbox`.
`<sf:checkboxes>`	Renders multiple HTML `<input>` tags with `type` set to `checkbox`.
`<sf:errors>`	Renders field errors in an HTML `<span>` tag.
`<sf:form>`	Renders an HTML `<form>` tag and exposed binding path to inner tags for data-binding.
`<sf:hidden>`	Renders an HTML `<input>` tag with `type` set to `hidden`.
`<sf:input>`	Renders an HTML `<input>` tag with `type` set to `text`.
`<sf:label>`	Renders an HTML `<label>` tag.
`<sf:option>`	Renders an HTML `<option>` tag. The `selected` attribute is set according to the `bound` value.
`<sf:options>`	Renders a list of HTML `<option>` tags corresponding to the bound collection, array, or map.
`<sf:password>`	Renders an HTML `<input>` tag with `type` set to `password`.
`<sf:radiobutton>`	Renders an HTML `<input>` tag with `type` set to `radio`.
`<sf:radiobuttons>`	Renders multiple HTML `<input>` tags with `type` set to `radio`.
`<sf:select>`	Renders an HTML `<select>` tag.
`<sf:textarea>`	Renders an HTML `<textarea>` tag.

Spring Form

Spring Form Tags

- <form:form commandName=“ “>
- <form:input path=“” />
- <form:checkbox path=“” value=“” />
- <form:radiobutton path=“” value=“” />
- <form:checkboxes path=“” items=“” />
- <form:radiobuttons path=“gender” items=“${genderOptions}” />
- <form:select path="skills" items="${skills}“/>
- <form:option value=“”>
- <form:options items=“” itemValue=“” itemLabel=“”/>
- <form:textarea path=“” rows=“” cols=“” />
- <form:hidden path=“”/>
- <form:errors path=“” />

  path can be *, field. If omitted then object errors are displayed.

Spring REST

- Spring uses message converters to convert data produced by control into a representation that is needed by client.
- Use @ResponseBody to inform Spring that we need to use message converter and send result directly to client. In other words, it skips normal model/view flow.
- Use @RequestBody to tell Spring to use message converter to convert data coming from client into an object.
- DispatcherServlet considers request’s ACCEPT header to determine the kind of content client wants.
- @RestController annotation applies message conversion to all methods in the controller. This was introduced in Spring 4.0

Spring HTTP Message Converters

- Message converters convert object to the required format.
- These message converters need extra libraries to be in classpath to work.

Jaxb2RootElementHttpMessageConverter
Reads and writes XML (either text/xml or application/xml) to and from JAXB2-annotated objects. *Registered if JAXB v2 libraries are present on the classpath.*

FormHttpMessageConverter
Reads content as application/x-www-form-urlencoded into a MultiValueMap<String,String>. Also writes MultiValueMap<String,String> as application/x-www-form-urlencoded and MultiValueMap<String, Object> as multipart/form-data.

MappingJacksonHttpMessageConverter Reads and writes JSON to and from typed objects or untyped HashMaps. *Registered if the Jackson JSON library is present on the classpath.*

Spring HTTP Message Converters

If the client's request has Accept header set to application/json then if Jackson JSON library is in the application's classpath, the object returned from the handler method is given to MappingJacksonHttpMessageConverter for conversion into a JSON representation to be returned to the client.

On the other hand, if the request header indicates that the client prefers text/xml, then Jaxb2RootElementHttpMessageConverter is tasked with producing an XML response to the client.

RestTemplate Methods

Method	Description
`delete()`	Performs an HTTP `DELETE` request on a resource at a specified URL
`exchange()`	Executes a specified HTTP method against a URL, returning a `ResponseEntity` containing an object mapped from the response body
`execute()`	Executes a specified HTTP method against a URL, returning an object mapped from the response body
`getForEntity()`	Sends an HTTP `GET` request, returning a `ResponseEntity` containing an object mapped from the response body
`getForObject()`	Sends an HTTP `GET` request, returning an object mapped from a response body
`headForHeaders()`	Sends an HTTP `HEAD` request, returning the HTTP headers for the specified resource URL
`optionsForAllow()`	Sends an HTTP `OPTIONS` request, returning the `Allow` header for the specified URL
`postForEntity()`	`POST`s data to a URL, returning a `ResponseEntity` containing an object mapped from the response body
`postForLocation()`	`POST`s data to a URL, returning the URL of the newly created resource
`postForObject()`	`POST`s data to a URL, returning an object mapped from the response

getForObject() Example

```
RestTemplate rest = new RestTemplate();
Book b = rest.getForObject
                    ("http://localhost:8888/webdemo/books/88", Book.class);
System.out.println(b.getTitle());
```

postForObject() Example

```
RestTemplate rest = new RestTemplate();
Book book = new Book("Java Comp. Ref.",1000);
rest.postForObject("http://localhost:8888/webdemo/books/add",
                               book, Book.class);
```