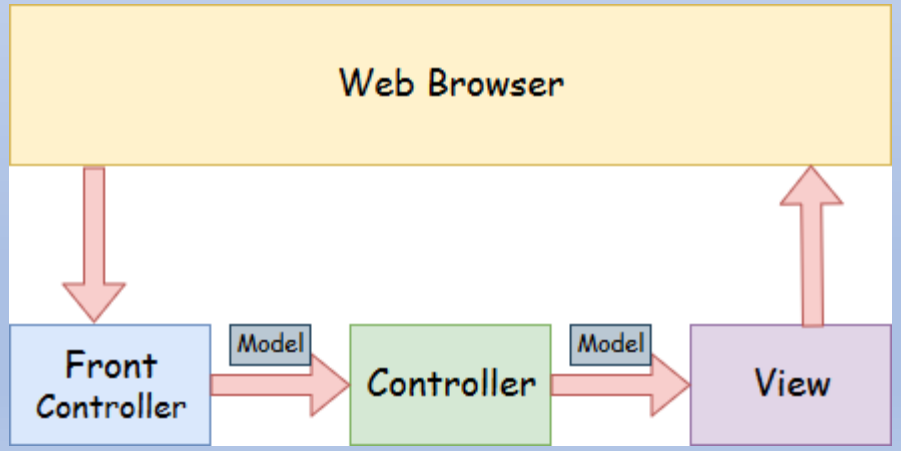
Spring Web MVC

Spring Web MVC

A Spring MVC is a Java framework which is used to build web applications. It follows the Model-View-Controller design pattern. It implements all the basic features of a core spring framework like Inversion of Control, Dependency Injection.

A Spring MVC provides an elegant solution to use MVC in spring framework by the help of DispatcherServlet. Here, DispatcherServlet is a class that receives the incoming request and maps it to the right resource such as controllers, models, and views.



Spring Web MVC

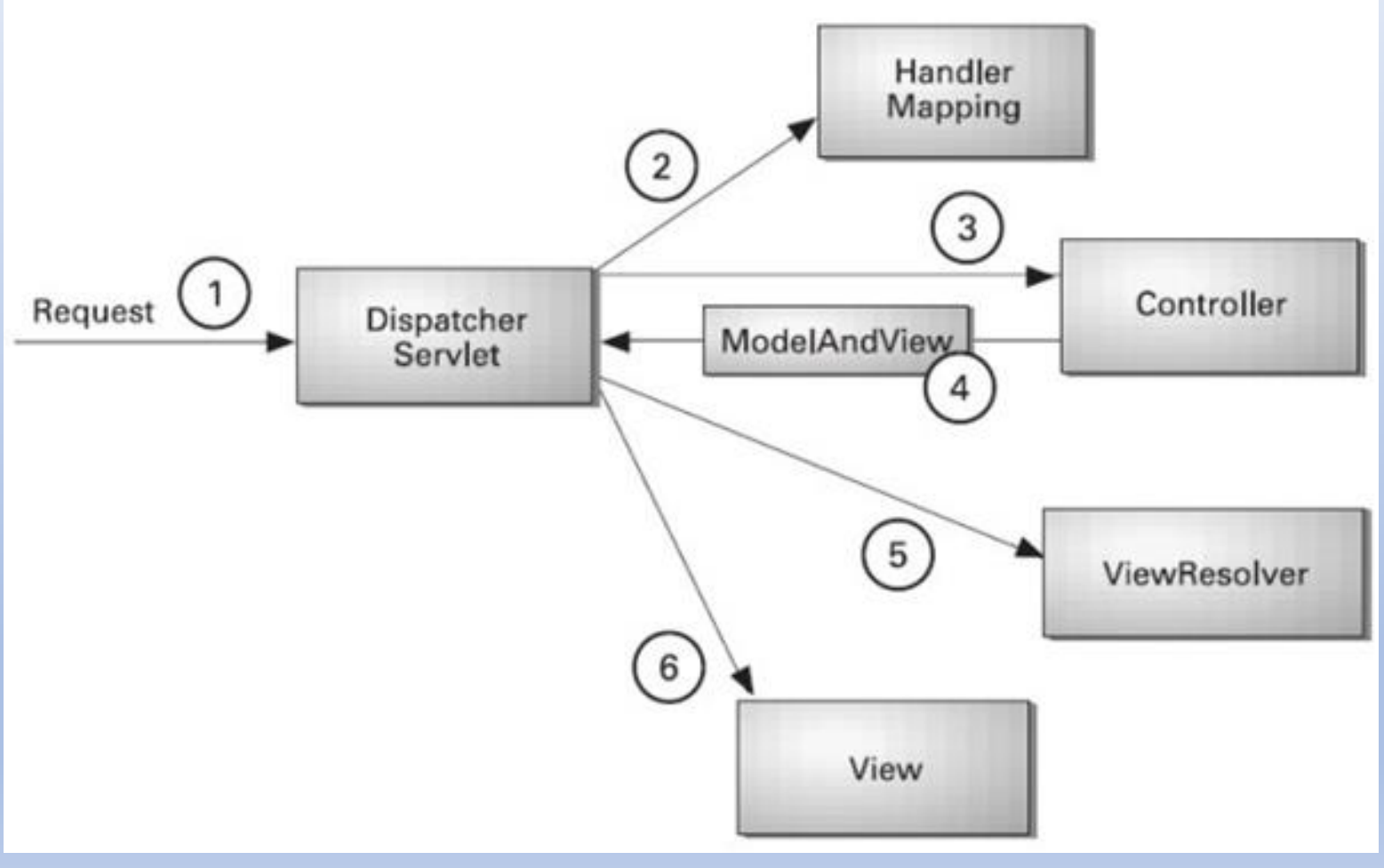
Model - A model contains the data of the application. A data can be a single object or a collection of objects.

Controller - A controller contains the business logic of an application. Here, the @Controller annotation is used to mark the class as the controller.

View - A view represents the provided information in a particular format. Generally, JSP+JSTL is used to create a view page. Although spring also supports other view technologies such as Apache Velocity, Thymeleaf and FreeMarker.

Front Controller - In Spring Web MVC, the DispatcherServlet class works as the front controller. It is responsible to manage the flow of the Spring MVC application.

Spring Web MVC Flow



Spring Web MVC

Advantages of Spring MVC Framework:

Separate roles - The Spring MVC separates each role, where the model object, controller, command object, view resolver, DispatcherServlet, validator, etc. can be fulfilled by a specialized object.

Light-weight - It uses light-weight servlet container to develop and deploy your application.

Powerful Configuration - It provides a robust configuration for both framework and application classes that includes easy referencing across contexts, such as from web controllers to business objects and validators.

Rapid development - The Spring MVC facilitates fast and parallel development.

Reusable business code - Instead of creating new objects, it allows us to use the existing business objects.

Easy to test - In Spring, generally we create JavaBeans classes that enable you to inject test data using the setter methods.

Flexible Mapping - It provides the specific annotations that easily redirect the page.

Spring Web MVC

Example:

```
@Controller
public class WelcomeController {

    @Autowired
    BookRepository bookRepository;

    @GetMapping({"/","/welcome"})
    public String welcome(@RequestParam(value = "name",
        defaultValue = "World", required = true) String name, Model model) {
        model.addAttribute("name", name);
        return "welcome";
    }

    @GetMapping({"/books"})
    public String books(Model model) {

        model.addAttribute("books", bookRepository.findAll());
        return "books/books";
```

Spring Boot Auto Configuration

- Spring Boot Auto Configuration and Dispatcher Servlet

- How does the Customer object gets converted to JSON?
- Who is configuring the error mapping?

Spring Boot Auto Configuration

Spring Boot automatically configures a spring application based on dependencies present or not present in the classpath as a jar, beans, properties, etc.

It makes development easier and faster as there is no need to define certain beans that are included in the auto-configuration classes.

A typical MVC database driven Spring MVC application requires a lot of configuration such as **dispatcher servlet, a view resolver, Jackson, data source, transaction manager,** among many others.

- Spring Boot auto-configures a **Dispatcher Servlet** if **Spring MVC jar** is on the classpath.
- Auto-configures the **Jackson** if **Jackson jar** is on the classpath.
- Auto-configures a **Data Source** if **Hibernate jar** is on the classpath.

Auto-configuration can be enabled by adding **@SpringBootApplication** or **@EnableAutoConfiguration** annotation in startup class. It indicates that it is a spring context file.

It enables something called **auto-configuration**.

It enable something called **components scan**. It is the features of Spring where it will start automatically scanning classes in the package and sub package for any bean file.

Spring Boot Auto Configuration

There is some example of auto configuration done by Spring Boot:

- **DispatcherServletAutoConfiguration**
- **DataSourceAutoConfiguration**
- **JacksonAutoConfiguration**
- **ErrorMvcAutoConfiguration** (#basicErrorController)

We can see the auto-configuration done by Spring Boot in the **AUTO-CONFIGURATION REPORT** or **CONDITIONS EVALUATION REPORT**.

Classes can be **excluded** from auto-configuration by adding:

@SpringBootApplication (exclude={JacksonAutoConfiguration.**class**, JmxAutoConfiguration.**class**})

Or add the following statement in the **application.properties** file.

spring.autoconfiguration.exclude=org.springframework.boot.autoconfigure.jackson.JacksonAutoConfiguration

We exclude classes form the auto-configuration for **faster startup** and **better performance** of the application.

AUTO-CONFIGURATION REPORT generated by enabling **debug** mode. Open the **application.properties** file and add the following statement:

logging.level.org.springframework=debug

Spring Boot Auto Configuration

Dispatcher Servlet

In Spring MVC all incoming requests go through a single servlet is called **Dispatcher Servlet (front controller)**. The front controller is a design pattern in web application development. A single servlet receives all the request and transfers them to all other components of the application.

The job of DispatcherServlet is to take an incoming URI and find the right combination of **handlers** (Controller classes) and **views** (usually JSPs). When the DispatcherServlet determines the view, it renders it as the response. Finally, the DispatcherServlet returns the Response Object to back to the client. In short, the Dispatcher Servlet plays the key role.

The other thing to notice is that ErrorMvcAutoConfiguration:

ErrorMvcAutoConfiguration matched:

-@ConditionalOnClass found required classes 'javax.servlet.Servlet',

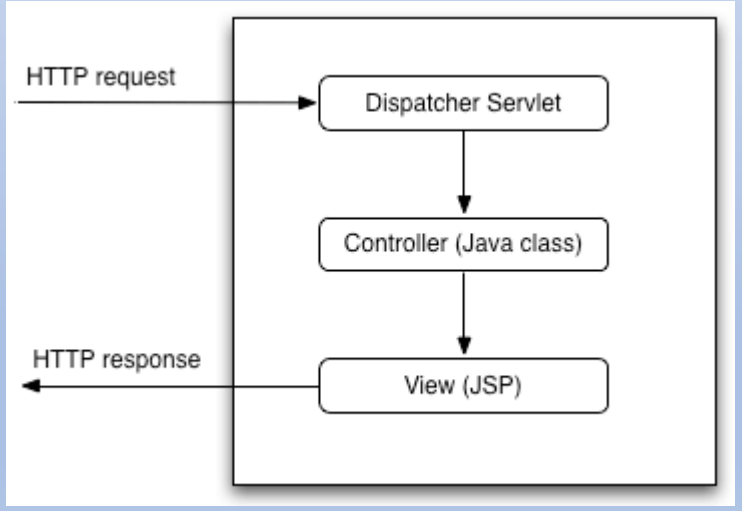
'org.springframework.web.servlet.DispatcherServlet' (OnClassCondition)- found

'session' scope (OnWebApplicatiossssnCondition)

It configures the **basicErrorController, errorAttributes, ErrorMvcAutoConfiguration,**

and **DefaultErrorViewResolverConfiguration**. It creates the default error page

which is known as **Whitelabel Error Page**.



Spring Boot Auto Configuration

It configures the **basicErrorController, errorAttributes,**

ErrorMvcAutoConfiguration,

and **DefaultErrorViewResolverConfiguration**. It creates the default error page

which is known as **Whitelabel Error Page**.

The other thing which is auto-configured

HttpMessageConvertersAutoConfiguration. *These message converter automatically converts the message.*

HttpMessageConvertersAutoConfiguration matched:

*-@ConditionalOnClass found required **class** 'org.springframework.http.converter.HttpMessageConverter' (OnClassCondition)*

----

*JacksonAutoConfiguration.Jackson2ObjectMapperBuilderCustomizerConfiguration matched: - @ConditionalOnClass found required **class** 'org.springframework.http.converter.json.Jackson2ObjectMapperBuilder'(OnClassCondition)*

It initializes the Jackson bean and the message converter. The **Jackson2ObjectMapper** does the conversion from **bean to JSON** and **JSON to bean**.

Spring MVC Annotations

- @RequestBody -
- @RequestMapping
- @GetMapping
- @PostMapping
- @PutMapping
- @DeleteMapping
- @PatchMapping
- @ControllerAdvice
- @ResponseBody
- @ExceptionHandler
- @ResponseStatus
- @PathVariable
- @RequestParam
- @Controller
- @RestController
- @ModelAttribute
- @CrossOrigin
- @InitBinder

Spring MVC Annotations

- **@RequestBody Annotation**

@RequestBody annotation indicating a method parameter should be bound to the body of the web request. The body of the request is passed through an *HttpMessageConverter* to resolve the method argument depending on the content type of the request. Optionally, automatic validation can be applied by annotating the argument with *@Valid*.

For example, the employee JSON object is converted into Java employee object using **@RequestBody** annotation.

```
@PostMapping("/users")
public User createUser(@Valid @RequestBody User user) {
return userRepository.save(user);
}
```

Spring automatically deserializes the JSON into a Java type assuming an appropriate one is specified. By default, the type we annotate with the @RequestBody annotation must correspond to the JSON sent from our client-side controller.

Spring MVC Annotations

- **@ResponseBody Annotation**

When you use the @ResponseBody annotation on a method, Spring converts the return value and writes it to the HTTP response automatically. Each method in the Controller class must be annotated with @ResponseBody.

- Spring has a list of *HttpMessageConverters* registered in the background. The responsibility of the *HTTPMessageConverter* is to convert the request body to a specific class and back to the response body again, depending on a predefined mime type. Every time an issued request hits *@ResponseBody*, Spring loops through all registered *HTTPMessageConverters* seeking the first that fits the given mime type and class and then uses it for the actual conversion.
- The @ResponseBody annotation tells a controller that the object returned is automatically serialized into JSON and passed back into the HttpResponse object.

```
@ResponseBody
@RequestMapping("/hello")
String hello() {
    return "Hello World!";
}
```

Spring MVC Annotations

- **@RequestMapping Annotation**

@RequestMapping annotation for mapping web requests onto methods in request-handling classes with flexible method signatures.

Both Spring MVC and Spring WebFlux support this annotation through a *RequestMappingHandlerMapping* and *RequestMappingHandlerAdapter* in their respective modules and package structure.

@RequestMapping marks request handler methods inside **@Controller** classes; it can be configured using:

path, or its aliases, name, and value: which URL the method is mapped to

method: compatible HTTP methods

params: filters requests based on the presence, absence, or value of HTTP parameters

headers: filters requests based on the presence, absence, or value of HTTP headers

consumes: which media types the method can consume in the HTTP request body

produces: which media types the method can produce in the HTTP response body Here's a quick example of what that looks like:

Spring MVC Annotations

- **@RequestMapping Annotation**

Example:

```
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
```

Spring MVC Annotations

- **@GetMapping Annotation**

@GetMapping annotation for mapping HTTP GET requests onto specific handler methods.

Specifically, **@GetMapping** is a composed annotation that acts as a shortcut for *@RequestMapping(method = RequestMethod.GET)*.

```
@GetMapping("/employees")
public List<Employee> getAllEmployees() {
return employeeRepository.findAll();
}
```

Spring MVC Annotations

- **@PostMapping Annotation**

@PostMapping annotation for mapping HTTP POST requests onto specific handler methods.

Specifically, **@PostMapping** is a composed annotation that acts as a shortcut for *@RequestMapping(method = RequestMethod.POST)*.

```
@PostMapping("/employees")
public Employee createEmployee(@Valid @RequestBody Employee employee) {
return employeeRepository.save(employee);
}
```

Spring MVC Annotations

- **@PutMapping Annotation**

@PutMapping annotation for mapping HTTP PUT requests onto specific handler methods.

Specifically, **@PutMapping** is a composed annotation that acts as a shortcut for *@RequestMapping(method = RequestMethod.PUT)*.

```
@PutMapping("/{customerId}")
public Customer updateCustomer(@PathVariable Long customerId, @RequestBody Customer customer) {
return customerService.updateCustomer(customer, customerId.intValue());
}
```

Spring MVC Annotations

- **@DeleteMapping Annotation**

@DeleteMapping annotation for mapping HTTP DELETE requests onto specific handler methods.

Specifically, **@DeleteMapping** is a composed annotation that acts as a shortcut for *@RequestMapping(method = RequestMethod.DELETE)*.

```
@DeleteMapping(value = "/customer/{id}")
public ResponseEntity<String> deleteCustomer(@PathVariable Long id) {
    customerService.deleteCustomer(id.intValue());
    return new ResponseEntity<>(HttpStatus.OK);
}
```

Spring MVC Annotations

- **@PatchMapping Annotation**

@PatchMapping annotation for mapping HTTP PATCH requests onto specific handler methods.

Specifically, **@PatchMapping** is a composed annotation that acts as a shortcut for *@RequestMapping(method = RequestMethod.PATCH)*.

```
@PatchMapping("/patch")
public @ResponseBody ResponseEntity<String> patch() {
    return new ResponseEntity<String>("PATCH Response", HttpStatus.OK);
}
```

Spring MVC Annotations

- **@ControllerAdvice Annotation**

@ControllerAdvice annotation is a specialization of *@Component*. The classes annotated with **@ControllerAdvice** are auto-detected by class path scanning.

The use of **@ControllerAdvice** is advising all or selected controllers for *@ExceptionHandler*, *@InitBinder,* and *@ModelAttribute*. What we have to do is create a class annotated with @ControllerAdvice and create a required method which will be annotated with @ExceptionHandler for **global exception handling**, *@InitBinder* for global init binding and *@ModelAttribute* for global model attributes addition. Whenever a request comes to a controller and its method with *@RequestMapping* and if there is no locally defined **@ExceptionHandler**, *@InitBinder* and *@ModelAttribute*, the globally defined class annotated with *@ControllerAdvice* is served.

```
@ControllerAdvice(basePackages = {"com.javaguides.springmvc.controller"} )
public class GlobalControllerAdvice {
```

Spring MVC Annotations

- **@ExceptionHandler Annotation**

@ExceptionHandler annotation for handling exceptions in specific handler classes and/or handler methods.

Handler methods which are annotated with this annotation are allowed to have very flexible signatures.

Spring calls this method when a request handler method throws any of the specified exceptions. The caught exception can be passed to the method as an argument:

```
@ExceptionHandler(PostNotFoundException.class)
public void handlePostNotFound(PostNotFoundException exception, HttpServletResponse response) throws
IOException{
response.sendError( HttpStatus.NOT_FOUND.value(), exception.getMessage() );
}
```

Spring MVC Annotations

- **@ResponseStatus Annotation**

We can specify the desired HTTP status of the response if we annotate a request handler method with this annotation. We can declare the status code with the code argument, or its alias, the value argument.

Also, we can provide a reason using the reason argument.

We also can use it along with **@ExceptionHandler**:

```
@ResponseStatus(HttpStatus.BAD_REQUEST)
@ExceptionHandler(ResourceNotFoundException.class)
public ResponseEntity<?> resourceNotFoundException(ResourceNotFoundException ex, WebRequest request) {
  ErrorDetails errorDetails = new ErrorDetails(new Date(), ex.getMessage(),
request.getDescription(false));
  return new ResponseEntity<>(errorDetails, HttpStatus.NOT_FOUND);
}
```

Spring MVC Annotations

- **@PathVariable Annotation**

This annotation indicates that a method argument is bound to a URI template variable. We can specify the URI template with the *@RequestMapping* annotation and bind a method argument to one of the template parts with *@PathVariable*.

We can achieve this with the name or its alias, the value argument:

```
@RequestMapping("/{id}")
public User getUser(@PathVariable("id") long id) {
    // ...
}
@RequestMapping("/{id}")
public User getUser(@PathVariable long id) {
    // ...
}
@RequestMapping("/{id}")
public User getUser(@PathVariable(required = false) long id) {
    // ...
}
```

Spring MVC Annotations

- **@RequestParam Annotation**

@RequestParam annotation which indicates that a method parameter should be bound to a web request parameter. We use **@RequestParam** for accessing HTTP request parameters:

```
@RequestMapping
Vehicle getVehicleByParam(@RequestParam("id") long id) {
    // ...
}

@RequestMapping("/buy")
Car buyCar(@RequestParam(defaultValue = "5") int seatCount) {
    // ...
}
```

Spring MVC Annotations

- **@Controller Annotation**

This annotation is simply a specialization of the *@Component* class and allows implementation classes to be autodetected through the classpath scanning.

We can define a Spring MVC controller with *@Controller*.

```
@Controller
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }
}
```

Spring MVC Annotations

- **@RestController Annotation**

Spring 4.0 introduced **@RestController**, a specialized version of the controller which is a convenience annotation that does nothing more than adding the **@Controller** and **@ResponseBody** annotations. By annotating the controller class with @RestController annotation, you no longer need to add **@ResponseBody** to all the request mapping methods. The @ResponseBody annotation is active by default.

```
@RestController
@RequestMapping("/api/v1")
public class EmployeeController {
    @Autowired
    private EmployeeRepository employeeRepository;

    @GetMapping("/employees")
    public List<Employee> getAllEmployees() {
        return employeeRepository.findAll();
    }
}
```

Spring MVC Annotations

- **@ModelAttribute Annotation**

@ModelAttribute annotation refers to the Model object in MVC. We can pass @ModelAttribute annotation to method arguments or we can even annotate a method as well.

Passing @ModelAttribute to method argument indicates that the value should bind to the method argument.

```
public String processForm(@ModelAttribute("person") Person person){
    person.getStuff();
}
```

Assume that we have a form that is backed by the Person object, when the user submits the form, the values should bind to the method argument with the help of @ModelAttribute annotation.

By annotating @ModelAttribute annotation to method defines the object, which will automatically be added to the Model, and later we can use the Model object inside the view template.

```
@ModelAttribute("person")
public Person getPerson(){
    return new Person();
}
```

Here the above method will allow access to the Person object in our View since it gets automatically gets added to the Models by Spring.

Note: Methods annotated with @ModelAttribute are invoked PRIOR TO every @RequestMapping method that is invoked in the same controller class. This is generally used for populating read-only components for a form – eg the values for a select list.

Spring MVC Annotations

- **@CrossOrigin Annotation**

@CrossOrigin enables cross-domain communication for the annotated request handler methods:

```
@CrossOrigin
@RequestMapping("/hello")
String hello() {
    return "Hello World!";
}
```

If we mark a class with it, it applies to all request handler methods in it.

We can fine-tune CORS behavior with this annotation’s arguments. This is usually required if you have an front end application developed in any of the JS library like (React, Vuejs, Angular etc) and wants to access SpringBoot backend app.

Spring MVC Annotations

- **@InitBinder Annotation**

@InitBinder annotation that identifies methods which initialize the WebDataBinder which will be used for populating command and form object arguments of annotated handler methods.

Such init-binder methods support all arguments that RequestMapping supports, except for command/form objects and corresponding validation result objects. Init-binder methods must not have a return value; they are usually declared as void.

```
// add an initbinder ... to convert trim input strings
    // remove leading and trailing whitespace
    // resolve issue for our validation
    @InitBinder
    public void initBinder(WebDataBinder dataBinder) {
        StringTrimmerEditor stringTrimmerEditor = new StringTrimmerEditor(true);
        dataBinder.registerCustomEditor(String.class, stringTrimmerEditor);
    }
```

Postman

The Collaboration Platform for API Development:

Postman is a popular API client that makes it easy for developers to create, share, test and document APIs. This is done by allowing users to create and save simple and complex HTTP/s requests, as well as read their responses. The result - more efficient and less tedious work

Web Service

Web services are the types of internet software that uses standardized messaging protocol over the distributed environment. It integrates the web-based application using the **REST, SOAP, WSDL,** and **UDDI** over the network. For example, Java web service can communicate with .Net application.

Features of web Services

- Web services are designed for application to application interaction.
- It should be interoperable.
- It should allow communication over the network.

Components of Web Services

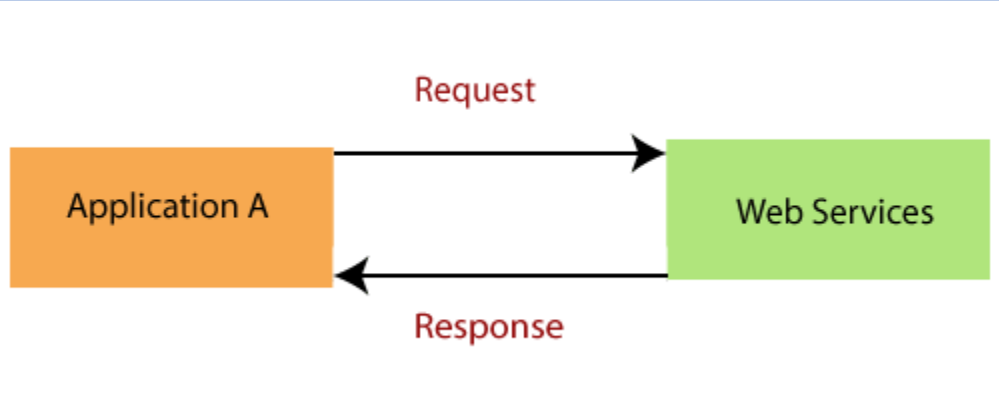
The web services must be able to fulfill the following conditions:

- The web service must be accessible over the internet.
- The web service is discoverable through a common mechanism like UDDI.
- It must be interoperable over any programming language or Operating System.

Web Service

Uses of Web Services

- Web services are used for reusing the code and connecting the existing program.
- Web services can be used to link data between two different platforms.
- It provides interoperability between disparate applications.
- How does data exchange between applications?
- Suppose, we have an **Application A** which create a request to access the **web services**. The web services offer a list of services. The web service process the **request** and sends the **response** to the Application A. The input to a web service is called a request, and the output from a web service is called response. The web services can be called from different platforms.



Web Service

There are two popular formats for request and response **XML** and **JSON**.

- **XML Format:** XML is the popular form as request and response in web services. Consider the following XML code:

```
<getDetail>
<id>DataStructureCourse</id>
</getDetail>
```

- The code shows that user has requested to access the DataStrutureCourse. The other data exchange format is JSON. JSON is supported by wide variety of platform.
- **JSON Format:** JSON is a readable format for structuring data. It is used for transiting data between server and web application.

```
{
   "employee":
  {
      "id": 00987
      "name":      "Jack",
      "salary":     20000,
  }
}
```

Web Service

To make a web service platform-independent, we make the **request** and **response** platform-independent.

Now a question arises, how does the **Application A** know the format of Request and Response?

The answer to this question is "Service Definition." Every web service offers a service definition. Service definition specifies the following:

- **Request/ Response format:** Defines the request format made by consumer and response format made by web service.
- **Request Structure:** Defines the structure of the request made by the application.
- **Response Structure:** Defines the structure of response returned by the web service.
- **Endpoint:** Defines where the services are available.

Web Service

Key Terminology of Web Services

- Request and Response
- Message Exchange Format: XML and JSON
- Service Provider or Server
- Service Consumer or Client
- Service Definition – WSDL, RAML, Swagger, Api Docs - Contract
- Transport: HTTP and MQ

Request and Response: Request is the input to a web service, and the response is the output from a web service.

Message Exchange Format: It is the format of the request and response. There are two popular message exchange formats: **XML** and **JSON**.

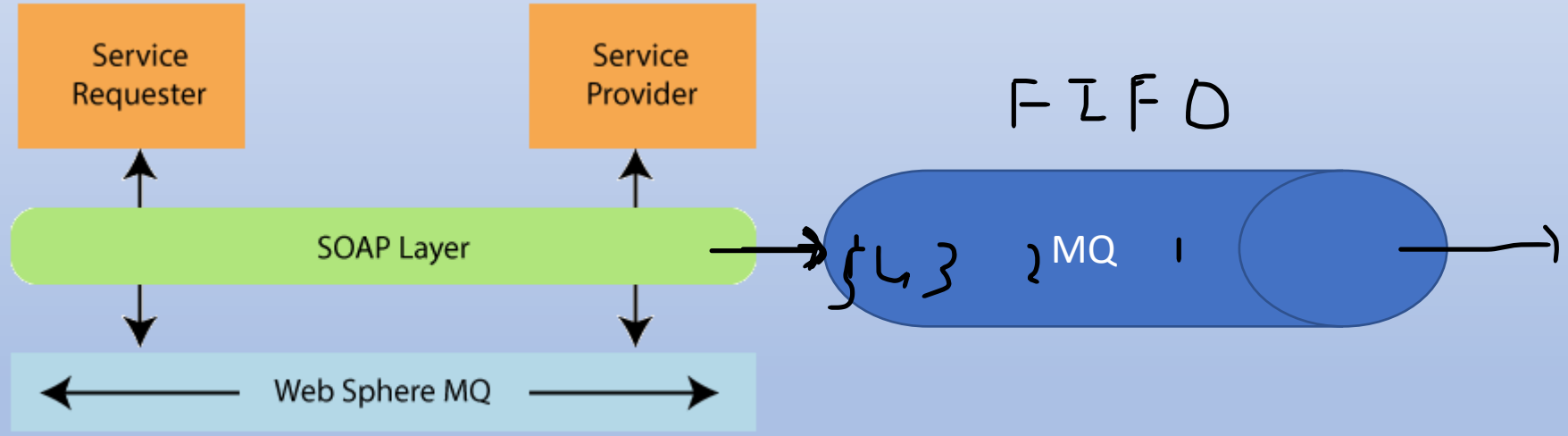
Service Provider or Server: Service provider is one which hosts the web service.

Service Consumer or Client: Service consumer is one who is using the web service.

Service Definition: Service definition is the **contract** between the service provider and service consumer. Service definition defines the format of request and response, request structure, response structure, and endpoint.

Web Service

Transport: Transport defines how a service is called. There is two popular way of calling a service: **HTTP** and Message Queue (**MQ**). By tying the URL of service, we can call the service over the internet. MQ communicates over the queue. The service requester puts the request in the queue. As soon as the service provider listens to the request. It takes the request, process the request, and create a response, and put the response back into MQ. The service requester gets the response from the queue. The communication happens over the queue.



How Service is called through Transport

Web Service

Characteristics of Web Services

- XML-based
- Coarse-grained
- Loosely coupled
- Capability to be synchronous and asynchronous
- Supports RPC

XML-based

A web service uses XML at information representation and record transportation layer. Using XML, there is no need of networking, operating system, or platform binding. Web offering based application is highly interoperable application at their middle level.

Coarse-grained

In the coarse-grained operation, a few objects hold a lot of related data. It provides broader functionality in comparison to fine-grained service. It wraps one or more fine-grained services together into a coarse-grained service. It is fine to have more coarse-grained service operations.

Loosely Coupled

A web service supports **loosely coupled** connections between systems. It communicates by passing XML message to each other via a web API. Web API adds a layer of abstraction to the environment that makes the connection adaptable and flexible.

Web Service

Capability to be synchronous and asynchronous

Synchronous Web services are invoked over existing Web protocols by a client who waits for a response. Synchronous Web services are served by **RPC-oriented messaging**.

Asynchronous Web services are invoked over existing Web protocols by a client who does not wait for a response. The **document-oriented messaging** often used for asynchronous Web services. Asynchronous Web Service is a crucial factor in enabling loosely coupled system.

Servlets, HTTP, and **XML/SOAP** are used to implement synchronous or asynchronous endpoints.

Supports RPC

A web service supports RPC through offering services of its personal, equivalent to those of a traditional aspect.

- A web service is a web resource. We can access a web service using platform-independent and language-neutral web protocols, such as HTTP. HTTP ensures easy integration of heterogeneous environment.
- A web service is typically registered. It can be located through a web service registry. A registry enables service consumers to find service that matches their needs. The service consumers may be human or other application.
- A web service provides an interface (a web API) that can be called from another program. The application-to-application programming can be invoked from any application.

Web Service

Architecture of Web Services

The Web Services architecture describes how to instantiate the elements and implement the operations in an interoperable manner.

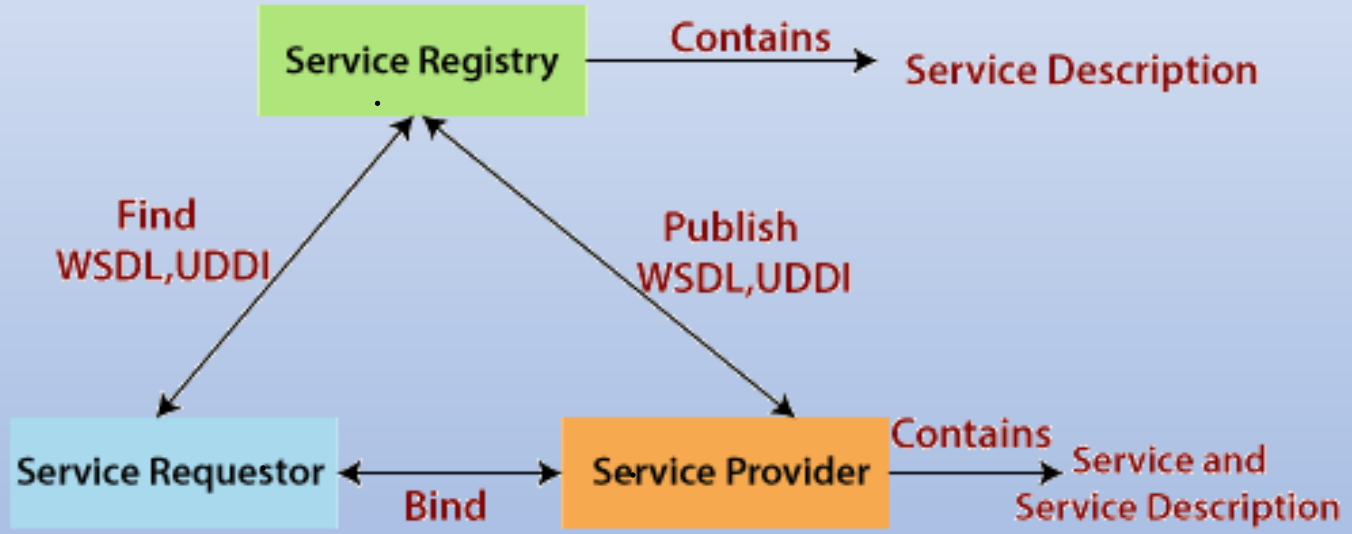
The architecture of web service interacts among three roles: **service provider**, **service requester**, and **service registry**. The interaction involves the three operations: **publish, find,** and **bind**. These operations and roles act upon the **web services artifacts**. The web service artifacts are the web service software module and its description.

The service provider hosts a network-associable module (web service). It defines a service description for the web service and publishes it to a service requestor or service registry. These service requestor uses a find operation to retrieve the service description locally or from the service registry. It uses the service description to bind with the service provider and invoke with the web service implementation.

The following figure illustrates the operations, roles, and their interaction.

Web Service

Architecture of Web Services



Web Service Roles, Operations and Artifacts

Web Service

Roles in a Web Service Architecture

There are three roles in web service architecture:

- Service Provider
- Service Requestor
- Service Registry

Service Provider

From an architectural perspective, it is the platform that hosts the services.

Service Requestor

Service requestor is the application that is looking for and invoking or initiating an interaction with a service. The browser plays the requester role, driven by a consumer or a program without a user interface.

Service Registry

Service requestors find service and obtain binding information for services during development.

Web Service

Operations in a Web Service Architecture

Three behaviors that take place in the microservices:

- Publication of service descriptions **(Publish)**
- Finding of services descriptions **(Find)**
- Invoking of service based on service descriptions **(Bind)**

Publish: In the publish operation, a service description must be published so that a service requester can find the service.

Find: In the find operation, the service requestor retrieves the service description directly. It can be involved in two different lifecycle phases for the service requestor:

At design, time to retrieve the service's interface description for program development.

And, at the runtime to retrieve the service's binding and location description for invocation.

Bind: In the bind operation, the service requestor invokes or initiates an interaction with the service at runtime using the binding details in the service description to locate, contact, and invoke the service.

Web Service

Artifacts of the web service

There are two artifacts of web services:

Service

Service Registry

Service: A service is an **interface** described by a service description. The service description is the implementation of the service. A service is a software module deployed on network-accessible platforms provided by the service provider. It interacts with a service requestor. Sometimes it also functions as a requestor, using other Web Services in its implementation.

Service Description: The service description comprises the details of the **interface** and **implementation** of the service. It includes its **data types, operations, binding information,** and **network location**. It can also categorize other metadata to enable discovery and utilize by service requestors. It can be published to a service requestor or a service registry.

Types of Web Services

There are two types of web services:

- RESTful Web Servies
- SOAP Web Services

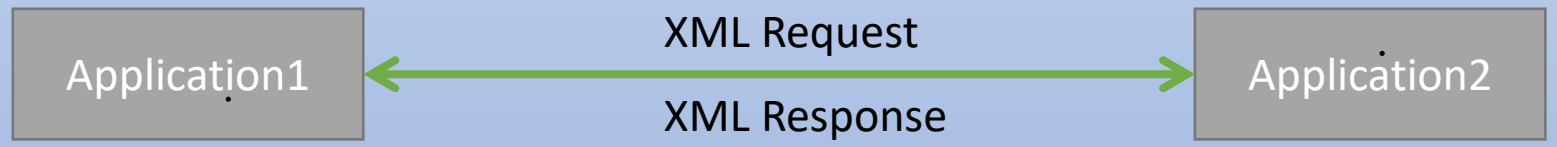
Web Service

SOAP Web Services

REST defines an architectural approach whereas SOAP poses a restriction on the format of the XML. XML transfer data between the service provider and service consumer. Remember that SOAP and REST are not **comparable**.

SOAP: SOAP acronym for **Simple Object Access Protocol**. It defines the standard XML format. It also defines the way of building web services. We use Web Service Definition Language (WSDL) to define the format of **request XML** and the **response XML**.

For example, we have requested to access the **Todo** application from the **Facebook** application. The Facebook application sends an XML request to the Todo application. Todo application processes the request and generates the XML response and sends back to the Facebook application.



Web Service

RESTful Web Services

REST stands for **REpresentational State Transfer**. It is developed by **Roy Thomas Fielding** who also developed HTTP. The main goal of RESTful web services is to make web services **more effective**. RESTful web services try to define services using the different concepts that are already present in HTTP. REST is an **architectural approach**, not a protocol.

It does not define the standard message exchange format. We can build REST services with both XML and JSON. JSON is more popular format with REST. The **key abstraction** is a resource in REST. A resource can be anything. It can be accessed through a **Uniform Resource Identifier (URI)**. For example:

The resource has representations like XML, HTML, and JSON. The current state is captured by representational resource. When we request a resource, we provide the representation of the resource. The important methods of HTTP are:

- **GET:** It reads a resource.
- **PUT:** It updates an existing resource.
- **POST:** It creates a new resource.
- **DELETE:** It deletes the resource.

Endpoint: localhost:8080/customers/121

Response:

“customerId”: 121

“customerName”: Amit

“address”: Pune

Web Service

For example, if we want to perform the following actions in the social media application, we get the corresponding results.

- **POST /users:** It creates a user.
- **GET /users/{id}:** It retrieve the detail of one user.
- **GET /users:** It retrieve the detail of all users.
- **DELETE /users:** It delete all users.
- **DELETE /users/{id}:** It delete a user.
- **GET /users/{id}/posts/post_id:** It retrieve the detail of a specific post.
- **POST / users/{id}/posts:** It creates a post for a user.

GET /users/{id}/posts: Retrieve all posts for a user

HTTP also defines the following standard status code:

- **404:** RESOURCE NOT FOUND
- **200:** SUCCESS
- **201:** CREATED
- **401:** UNAUTHORIZED
- **500:** SERVER ERROR

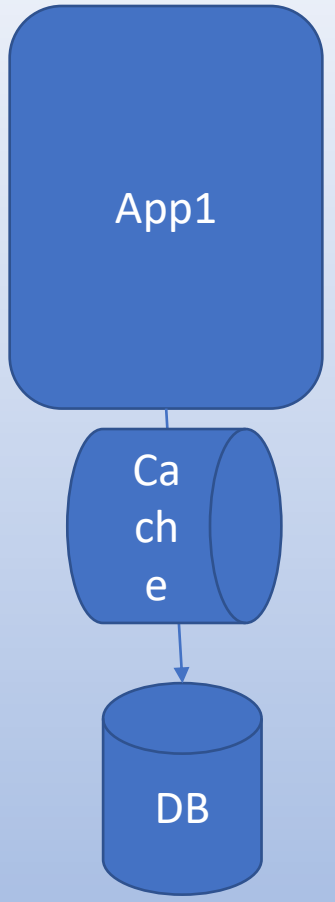
Web Service

RESTful Service Constraints

- There must be a service producer and service consumer.
- The service is stateless.
- The service result must be cacheable.
- The interface is uniform and exposing resources.
- The service should assume a layered architecture.

Advantages of RESTful web services

- RESTful web services are **platform-independent**.
- It can be written in any programming language and can be executed on any platform.
- It provides different data format like **JSON, text, HTML,** and **XML**.
- It is fast in comparison to SOAP because there is no strict specification like SOAP.
- These are **reusable**.
- These are **language neutral**.



Web Service

Difference:

SOAP Protocol	RESTful Web Services
SOAP is a protocol.	REST is an architectural approach.
SOAP acronym for Simple Object Access Protocol.	REST acronym for REpresentational State Transfer.
In SOAP, the data exchange format is always XML.	There is no strict data exchange format. We can use JSON, XML, etc.
XML is the most popular data exchange format in SOAP web services.	JSON is the most popular data exchange format in RESTful web services.
SOAP uses Web Service Definition Language (WSDL).	REST does not have any standard definition language.
SOAP does not pose any restrictions on transport. We can use either HTTP or MQ.	RESTful services use the most popular HTTP protocol.
SOAP web services are typical to implement.	RESTful services are easier to implement than SOAP.
SOAP web services use the JAX-WS API.	RESTful web services use the JAX-RS API.
SOAP protocol defines too many standards.	RESTful services do not emphasis on too many standards.
SOAP cannot use RESTful services because it is a protocol.	RESTful service can use SOAP web services because it is an architectural approach that can use any protocol like HTTP and SOAP.
SOAP reads cannot be cached.	REST reads can be cached.

REST API

REST is acronym for Representational State Transfer. It is architectural style for distributed hypermedia systems and was first presented by Roy Fielding in 2000 in his famous dissertation. Like any other architectural style, REST also does have it's own 6 guiding constraints which must be satisfied if an interface needs to be referred as **RESTful**. These principles are listed below:

REST API

Guiding Principles of REST:

1. **Client–server** – By separating the user interface concerns from the data storage concerns, we improve the portability of the user interface across multiple platforms and improve scalability by simplifying the server components.
2. **Stateless** – Each request from client to server must contain all of the information necessary to understand the request, and cannot take advantage of any stored context on the server. Session state is therefore kept entirely on the client.
3. **Cacheable** – Cache constraints require that the data within a response to a request be implicitly or explicitly labeled as cacheable or non-cacheable. If a response is cacheable, then a client cache is given the right to reuse that response data for later, equivalent requests.
4. **Uniform interface** – By applying the software engineering principle of generality to the component interface, the overall system architecture is simplified and the visibility of interactions is improved. In order to obtain a uniform interface, multiple architectural constraints are needed to guide the behavior of components. REST is defined by four interface constraints: identification of resources; manipulation of resources through representations; self-descriptive messages; and, hypermedia as the engine of application state.
5. **Layered system** – The layered system style allows an architecture to be composed of hierarchical layers by constraining component behavior such that each component cannot “see” beyond the immediate layer with which they are interacting.
6. **Code on demand (optional)** – REST allows client functionality to be extended by downloading and executing code in the form of applets or scripts. This simplifies clients by reducing the number of features required to be pre-implemented

REST API

Resource:

The key abstraction of information in REST is a resource. Any information that can be named can be a resource: a document or image, a temporal service, a collection of other resources, a non-virtual object (e.g. a person), and so on. REST uses a resource identifier to identify the particular resource involved in an interaction between components.

The state of the resource at any particular timestamp is known as **resource representation**. A representation consists of data, metadata describing the data and hypermedia links which can help the clients in transition to the next desired state.

REST API

REST and HTTP are not same !!

In the REST architectural style, data and functionality are considered resources and are accessed using Uniform Resource Identifiers (URIs). The resources are acted upon by using a set of simple, well-defined operations. The clients and servers exchange representations of resources by using a standardized interface and protocol – typically HTTP.

REST API

Resource Methods:

Another important thing associated with REST is resource methods to be used to perform the desired transition. A large number of people wrongly relate resource methods to HTTP GET/PUT/POST/DELETE methods.

Roy Fielding has never mentioned any recommendation around which method to be used in which condition. All he emphasizes is that it should be uniform interface. If you decide HTTP POST will be used for updating a resource – rather than most people recommend HTTP PUT – it's alright and application interface will be RESTful.

REST API

REST Resource Naming Guide

- A resource can be a singleton or a collection - /customers
- A resource may contain sub-collection resources, example

  /customers/{customerId}/accounts/{accountId}

REST APIs use Uniform Resource Identifiers (URIs) to address resources. REST API designers should create URIs that convey a REST API's resource model to its potential client developers. When resources are named well, an API is intuitive and easy to use.

REST API

REST Resource Naming Best Practices

- Use nouns to represent resources

  http://api.example.com/device-management/managed-devices

- Consistency is the key

  http://api.example.com/device-management/managed-devices/{id}/scripts/{id}

- Never use CRUD function names in URIs

  HTTP POST http://api.example.com/device-management/managed-devices //Create new Device

- Use query component to filter URI collection

  http://api.example.com/device-management/managed-devices?region=USA&brand=XYZ

REST API

Http Methods:

- HTTP GET - to retrieve resource representation/information only
- HTTP POST - to create new subordinate resources
- HTTP PUT - to update existing resource
- HTTP DELETE - to delete resources
- HTTP PATCH - to make partial update on a resource
- Note: [HTTP Methods – REST API Verbs (restfulapi.net)]

REST API

HTTP METHOD	CRUD	ENTIRE COLLECTION (E.G. /USERS)	SPECIFIC ITEM (E.G. /USERS/123)
POST	Create	201 (Created), 'Location' header with link to /users/{id} containing new ID.	Avoid using POST on single resource
GET	Read	200 (OK), list of users. Use pagination, sorting and filtering to navigate big lists.	200 (OK), single user. 404 (Not Found), if ID not found or invalid.
PUT	Update/Replace	405 (Method not allowed), unless you want to update every resource in the entire collection of resource.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
PATCH	Partial Update/Modify	405 (Method not allowed), unless you want to modify the collection itself.	200 (OK) or 204 (No Content). Use 404 (Not Found), if ID not found or invalid.
DELETE	Delete	405 (Method not allowed), unless you want to delete the whole collection — use with caution.	200 (OK). 404 (Not Found), if ID not found or invalid.

REST API

Http Status Codes:

- HTTP defines these standard status codes that can be used to convey the results of a client's request. The status codes are divided into the five categories.
- 1xx: Informational – Communicates transfer protocol-level information.
- 2xx: Success – Indicates that the client's request was accepted successfully.
- 3xx: Redirection – Indicates that the client must take some additional action in order to complete their request.
- 4xx: Client Error – This category of error status codes points the finger at clients.
- 5xx: Server Error – The server takes responsibility for these error status codes.

REST API Caching

Caching:

- An architectural constraints
- GET requests should be cachable by default – until a special condition arises. Usually, browsers treat all GET requests as cacheable.
- POST requests are not cacheable by default but can be made cacheable if either an Expires header or a Cache-Control header with a directive, to explicitly allows caching, is added to the response.
- Responses to PUT and DELETE requests are not cacheable at all.

REST API Caching

Below given are main HTTP response headers that we can use to control caching behavior:

- **Expires**

The Expires HTTP header specifies an absolute expiry time for a cached representation. Beyond that time, a cached representation is considered stale and must be re-validated with the origin server. To indicate that a representation never expires, a service can include a time up to one year in the future.

Expires: Fri, 20 May 2016 19:20:49 IST

- **Cache-Control**

The header value comprises one or more comma-separated directives. These directives determine whether a response is cacheable, and if so, by whom, and for how long e.g. max-age or s-maxage directives.

Cache-Control: max-age=3600

Cacheable responses (whether to a GET or to a POST request) should also include a validator — either an ETag or a Last-Modified header.

REST API Caching

- **ETag**

An ETag value is an opaque string token that a server associates with a resource to uniquely identify the state of the resource over its lifetime. When the resource changes, the ETag changes accordingly.

ETag: "abcd1234567n34jv"

- **Last-Modified**

Whereas a response's Date header indicates when the response was generated, the Last-Modified header indicates when the associated resource last changed. The Last-Modified value cannot be less than the Date value.

Last-Modified: Fri, 10 May 2016 09:17:49 IST

REST API Content Negotiation

Most of the REST API implementations rely on agent-driven content negotiations. Agent-driven content negotiation depends on the usage of HTTP request headers or resource URI patterns.

- **Using HTTP Headers**

At server side, an incoming request may have an entity attached to it. To determine it's type, server uses the HTTP request header **Content-Type**. Some common examples of content types are "text/plain", "application/xml", "text/html", "application/json", "image/gif", and "image/jpeg".

Content-Type: application/json

Similarly, to determine what type of representation is desired on the client-side, an HTTP header ACCEPT is used. It will have one of the values mentioned for Content-Type above.

Accept: application/json

REST API Content Negotiation

Generally, if no Accept header is present in the request, the server can send pre-configured default representation type.

- **Using URL Patterns**

Another way to pass content type information to the server, the client may use the specific extension in resource URIs.

http://rest.api.com/v1/employees/20423.xml

http://rest.api.com/v1/employees/20423.json

the first request URI will return an XML response whether the second request URI will return a JSON response.

REST API Idempotent

An idempotent HTTP method is a method that can be invoked many times without the different outcomes. It should not matter if the method has been called only once, or ten times over. The result should always be the same.

Idempotency essentially means that the result of a successfully performed request is independent of the number of times it is executed.

- **Http Methods**

If we follow the REST principles in designing our APIs, we will have automatically idempotent REST APIs for GET, PUT, DELETE, HEAD, OPTIONS, and TRACE methods. Only POST APIs will not be idempotent.

- POST is NOT idempotent.
- GET, PUT, DELETE, HEAD, OPTIONS and TRACE are idempotent.

REST API Idempotent

- **Http Post**

when we invoke the same POST request N times, we will have N new resources on the server. So, POST is not idempotent.

POST is NOT idempotent.

- **Http GET, PUT, DELETE, HEAD, OPTIONS and TRACE**

GET, HEAD, OPTIONS and TRACE methods NEVER change the resource state on the server. They are purely for retrieving the resource representation or metadata at that point in time.

PUT will update the same resource for multiple requests

DELETE will delete the resource on first request and remaining n-1 requests throw 404.

REST API Security

REST Security Design Principles

- **Least Privilege**: An entity should only have the required set of permissions to perform the actions for which they are authorized, and no more. Permissions can be added as needed and should be revoked when no longer in use.
- **Fail-Safe Defaults**: A user's default access level to any resource in the system should be "denied" unless they've been granted a "permit" explicitly.
- **The economy of Mechanism**: The design should be as simple as possible. All the component interfaces and the interactions between them should be simple enough to understand.
- **Complete Mediation**: A system should validate access rights to all its resources to ensure that they're allowed and should not rely on the **cached permission matrix**. If the access level to a given resource is being revoked, but that isn't reflected in the permission matrix, it would violate the security.

REST API Security

REST Security Design Principles

- **Open Design**: This principle highlights the importance of building a system in an open manner—with no secret, confidential algorithms.
- **Separation of Privilege**: Granting permissions to an entity should not be purely based on a single condition, a combination of conditions based on the type of resource is a better idea.
- **Least Common Mechanism**: It concerns the risk of sharing state among different components. If one can corrupt the shared state, it can then corrupt all the other components that depend on it.
- **Psychological Acceptability**: It states that security mechanisms should not make the resource more difficult to access than if the security mechanisms were not present. In short, security should not make worse the user experience.

REST API Security

Best Practices to Secure REST APIs

- **Always Use HTTPS**
- **Use Password Hash**
- **Never expose information on URLs**
- **Consider Oauth**
- **Input Parameter Validation**

REST API Versioning

APIs only need to be up-versioned when a breaking change is made.

Ways to version Rest Api:

- **URI Versioning:**

  We can include dates, project names or any other meaningful identifier.

  http://api.example.com/v1

- **Custom Request Header:**

  Accept-version: v1

- **Accept header:**

  Accept: application/vnd.example.v1+json

  Accept: application/vnd.example+json;version=1.0

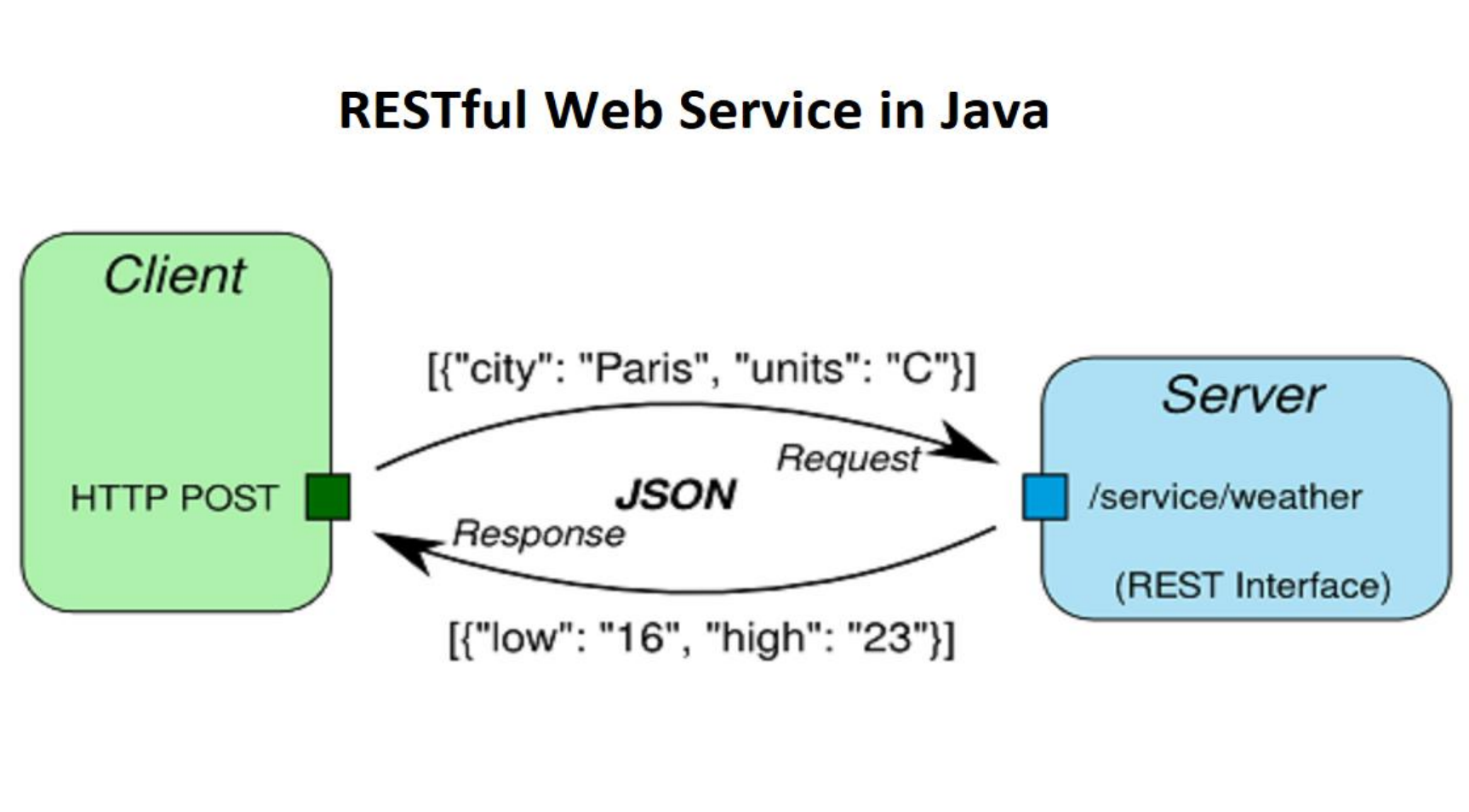
Spring REST

Spring Boot provides a very good support to building RESTful Web Services for enterprise applications.

Dependency:

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

Spring REST


RESTful Web Service in Java
Client
HTTP POST
[{"city": "Paris", "units": "C"}]
Request
JSON
Response
[{"low": "16", "high": "23"}]
Server
/service/weather
(REST Interface)

Content Negotiation using Spring Boot

Usual scenarios we specify if a method should return a response either as xml,json,html or some other type. But it may happen that we need the same method to return a response of different type depending on the incoming request.

For example we have a Server application which return the list of employees. We are having 2 UI applications using the same server method to get the list of employees. But one requires xml and the other json.

This is where content negotiation comes into picture. As name suggests it negotiates the response type based on the request. This content negotiation can be achieved in following ways-

- **Using Path Extension** - In the request we specify the required response type using the extension like .json,.xml or .txt. This has the highest preference but not recommended for REST api development in Spring.
- **Using URL parameter** - In the request we specify the required response type using the url parameter like format=xml or format=json. This has the second highest preference
- **Using Accept Headers** - When making a request using HTTP we specify required response by setting the Accept header property. Its something like this

```
HttpHeaders headers = new HttpHeaders();

headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
```

Content Negotiation using Spring Boot

There are 2 ways for generating output in our Spring MVC application.

- For RESTful API, use @ResponseBody annotation. Spring MVC HTTP message converters will return data in the required format (e.g. JSON, XML etc.).
- For the traditional applications, viewResolver used to generate presentation format like HTML.
- There is a third possibility which requires both RESTful and traditional web-based data.

For above use cases, it’s desire to know what kind of data format expected in the request body and what it expects in the HTTP response. Spring MVC uses **ContentNegotationStrategy** to determine what format requested by the user.

Content Negotiation using Spring Boot

Example:

```
@Configuration
@EnableWebMvc
public class WebConfig implements WebMvcConfigurer {
@Override
public void configureContentNegotiation(ContentNegotiationConfigurer configurer) {
     configurer.
     //for url parameter - query string
     favorParameter(true).
     parameterName("mediaType").
     //accept header -> mark as false
     ignoreAcceptHeader(false).
     defaultContentType(MediaType.APPLICATION_XML).
     mediaType("xml", MediaType.APPLICATION_XML)
     .mediaType("json", MediaType.APPLICATION_JSON);
     }
}
```

Api Version with Spring Boot

There are some different ways to provide API versioning in your application. The most popular of them are:

1. Through a **URI path** – you include the version number in the URL path of the endpoint, for example, */api/v1/persons.*
2. Through **query parameters** – you pass the version number as a query parameter with a specified name, for example, */api/persons?version=1.*
3. Through **custom HTTP headers** – you define a new header that contains the version number in the request.
4. Through a **content negotiation** – the version number is included in the "Accept" header together with the accepted content type. The request with cURL would look like the following:

```
curl -H "Accept: application/vnd.piomin.v1+json" http://localhost:8080/api/persons
```

Spring boot Embedded server

An embedded server is embedded as part of the deployable application.

If we talk about Java applications, that would be a JAR.

The advantage with this is you don’t need the server; pre-installed in the deployment environment.

With SpringBoot, the default embedded server is Tomcat. Other options available are Jetty and UnderTow.

Spring boot Embedded server

Embedded servers are quite scalable, and can host applications that support millions of users. These are no less scalable than the conventional fat servers.

As a concept, embedded servers might take some time to get used to. These can be used with applications for deployment in high-workload environments, without sacrificing any reliability or stability.

Embedded servers are also quite lightweight. If you look at a conventional WebSphere or Weblogic installation, or even a default Tomcat setup, their install sizes are huge!

Embedded server images don’t generally result in huge archive sizes and helps in building smaller containers.

Use Another Web Server

spring-boot-starter-web includes Tomcat by including spring-boot-starter-tomcat, but we can use spring-boot-starter-jetty or spring-boot-starter-undertow instead.

By default, the Spring Boot framework uses Tomcat as the embedded server of choice. However, you could override this default setting by specifying certain configuration settings. For instance, if you want to use a Jetty dependency instead, then use an <exclusion> element in the XML configuration file, and specify a <dependency> element as well:

Use Another Web Server

```
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-web</artifactId>
        <exclusions>
                <!-- Exclude the Tomcat dependency -->
                <exclusion>
                        <groupId>org.springframework.boot</groupId>
                        <artifactId>spring-boot-starter-tomcat</artifactId>
                </exclusion>
        </exclusions>
</dependency>
<!-- Use Jetty instead -->
<dependency>
        <groupId>org.springframework.boot</groupId>
        <artifactId>spring-boot-starter-jetty</artifactId>
</dependency>
```

Change the HTTP Port

In a standalone application, the main HTTP port defaults to 8080 but can be set with server.port (for example, in application.properties or as a System property). Thanks to relaxed binding of Environment values, you can also use SERVER_PORT (for example, as an OS environment variable).

server.port=8083

Spring MVC, JAX-RS and Jeresy

JAX-RS is a specification for implementing REST web services in Java, currently defined by the JSR-370. It is part of the Java EE technologies, currently defined by the JSR 366.

Jersey (shipped with GlassFish and Payara) is the JAX-RS reference implementation, however there are other implementations such as RESTEasy (shipped with JBoss EAP and WildFly) and Apache CXF (shipped with TomEE and WebSphere).

The Spring Framework is a full framework that allows you to create Java enterprise applications. The REST capabilities are provided by the Spring MVC module (same module that provides model-view-controller capabilities). It is not a JAX-RS implementation and can be seen as a Spring alternative to the JAX-RS standard.

The Spring ecosystem also provides a wide range of projects for creating enterprise applications, covering persistence, security, integration with social networks, batch processing, etc.

Spring MVC, JAX-RS and Jeresy

Spring to JAX-RS Cheat Sheet

This is not an exhaustive list, but it does include the most common annotations.

Spring Annotation	JAX-RS Annotation
`@RequestMapping(path = "/troopers"`	`@Path("/troopers")`
`@RequestMapping(method = RequestMethod.POST)`	`@POST`
`@RequestMapping(method = RequestMethod.GET)`	`@GET`
`@RequestMapping(method = RequestMethod.DELETE)`	`@DELETE`
`@ResponseBody`	N/A
`@RequestBody`	N/A
`@PathVariable("id")`	`@PathParam("id")`
`@RequestParam("xyz")`	`@QueryParam('xyz")`
`@RequestParam(value="xyz"`	@FormParam("xyz")
`@RequestMapping(produces = {"application/json"})`	`@Produces("application/json")`
`@RequestMapping(consumes = {"application/json"})`	`@Consumes("application/json")`

Spring Boot Thymeleaf

Thymeleaf is a Java-based library used to create a web application. It provides a good support for serving a XHTML/HTML5 in web applications.

Thymeleaf Templates

Thymeleaf converts your files into well-formed XML files. It contains 6 types of templates as given below −

XML
Valid XML
XHTML
Valid XHTML
HTML5
Legacy HTML5

All templates, except Legacy HTML5, are referring to well-formed valid XML files. Legacy HTML5 allows us to render the HTML5 tags in web page including not closed tags.

Spring Boot Thymeleaf

We can use Thymeleaf templates to create a web application in Spring Boot. You will have to follow the below steps to create a web application in Spring Boot by using Thymeleaf.

Below code to create a @Controller class file to redirect the Request URI to HTML file.

```
@RestController
public class WebController {
   @RequestMapping(value = "/index")
   public String index() {
      return "index";
   }
}
```

Spring Boot Thymeleaf

add the following dependency into the pom.xml file:

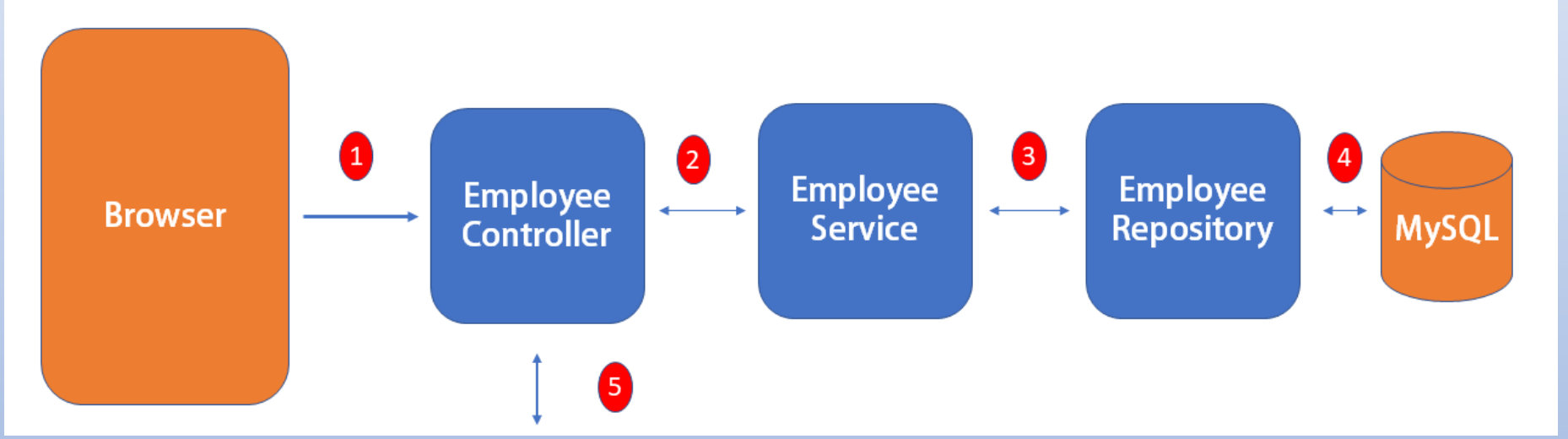
```
<dependency>
   <groupId>org.springframework.boot</groupId>
   <artifactId>spring-boot-starter-thymeleaf</artifactId>
</dependency>
```

add the following dependency in the build.gradle file:

```
implementation 'org.springframework.boot:spring-boot-starter-thymeleaf'
```

Spring Boot Thymeleaf

- A sample of employee CRUD application architecture



Spring Validations

Validating the user inputs is one of the common tasks that the developer should implement during web application development. Validation may include checking the correct mobile number format, the minimum size of an input, etc.

The spring-boot-starter-web dependency is bundled with spring-boot-starter-validation starter dependency. This dependency contains the Bean validation API, that is used for form validation.

Spring MVC Validations Example

```
@Data
public class Student {
@NotNull(message = "LastName can not be null!!")
@NotEmpty(message = "LastName can not be empty!!")
private String name;
@NotNull(message = "Choose the subject count you are going to study!")
@Min(value = 4, message = "Student should enroll to minimum 4 subjects!!")
@Max(value = 8, message = "Student can enroll to maximum 8 subjects!!")
private int subjectCount;
@NotNull
@Min(1)
@Max(12)
private int grade;
```