Mastering the Spring Framework

CHAPTER 4:
REST WEB SERVICES WITH SPRING BOOT

Chapter Objectives

In this chapter, we will:

- Introduce REST Web Services
- Introduce Spring Boot
- Learn how to write REST Web services using Spring Boot

Chapter Concepts

Web Services

HTTP and JSON

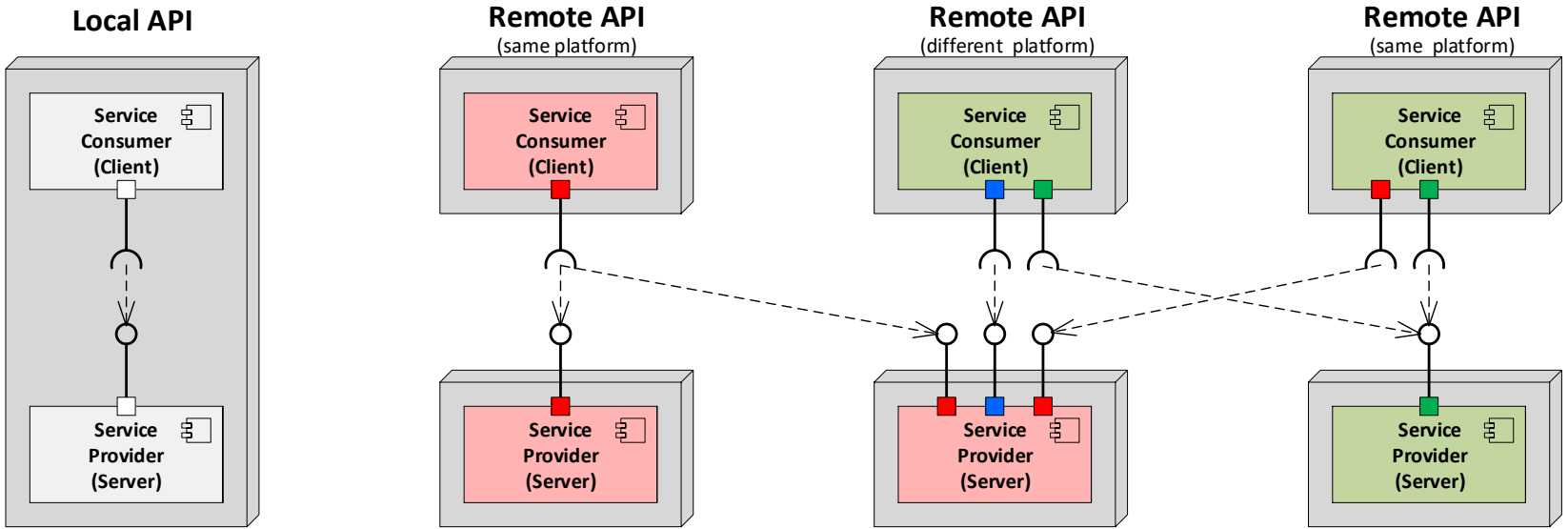
RESTful Services

Spring Boot

Exercise

Application Programming Interface (API)

- API is a set of clearly defined methods of **communication** between various software components



Web Services

- Web Service
  - Cross-platform way to integrate applications
  - Application functionality exposed over network, typically over WWW
  - Communication protocols: usually HTTP, but can use other protocols such as ESMTP, message queues, etc.
- They provide great interoperability and extensibility
- They are loosely coupled
  - Can be combined to build complex applications
  - Components can be developed in different languages on different architectures

Types of Web Services

- **S**imple **O**bject **A**ccess **P**rotocol (SOAP) Web Services
  - Interfaces defined using Web Services Description Language (WSDL)
  - Messages are exchanged in XML
- **Re**presentational **S**tate **T**ransfer (RESTful) Web Services
  - Lightweight infrastructure which is completely stateless
  - Implementations require minimal tooling

SOAP vs. REST: Typical Use Cases

- **S**imple **O**bject **A**ccess **P**rotocol (SOAP) Web Services
  - RPC style of integration (**verb-first**)
  - System to System integration within a single enterprise or across enterprises
  - Presence or need for enterprise-wide integration standards (primarily WS-Security)
  - Strong formal service contracts and formal governance (in most cases)
  - Service consumers are known and very often formal agreements
- **Re**presentational **S**tate **T**ransfer (RESTful) Web Services
  - 'Document' CRUD style of integration (**noun-first**)
  - Client (Browser) to System as well as System to System integration
  - Many 'unknown' consumers (client apps for Yahoo, Google, etc.—any Internet service)
  - Need for high adaptability and flexibility

REST – High-Level Overview

- **Re**presentative **S**tate **T**ransfer—REST or ReST
  - A software architecture style
  - Guidelines and best practices for creating scalable web services
  - Described in Roy Fielding's doctoral thesis
- Typically communicates over HTTP
- Common data exchange format – JSON
- REST was developed by W3C in parallel with HTTP 1.1
  - The World Wide Web is an implementation of REST
- There is no official standard for REST APIs
  - REST is an architectural style
  - SOAP is a protocol which has standards
  - REST usually uses standards such as HTTP, URI, JSON, XML

Chapter Concepts

Web Services

HTTP and JSON

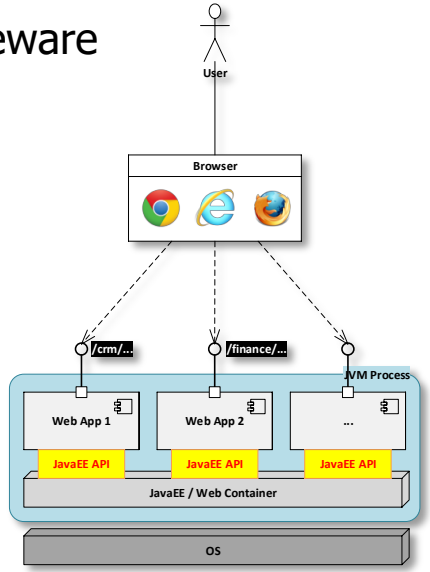
RESTful Services

Spring Boot

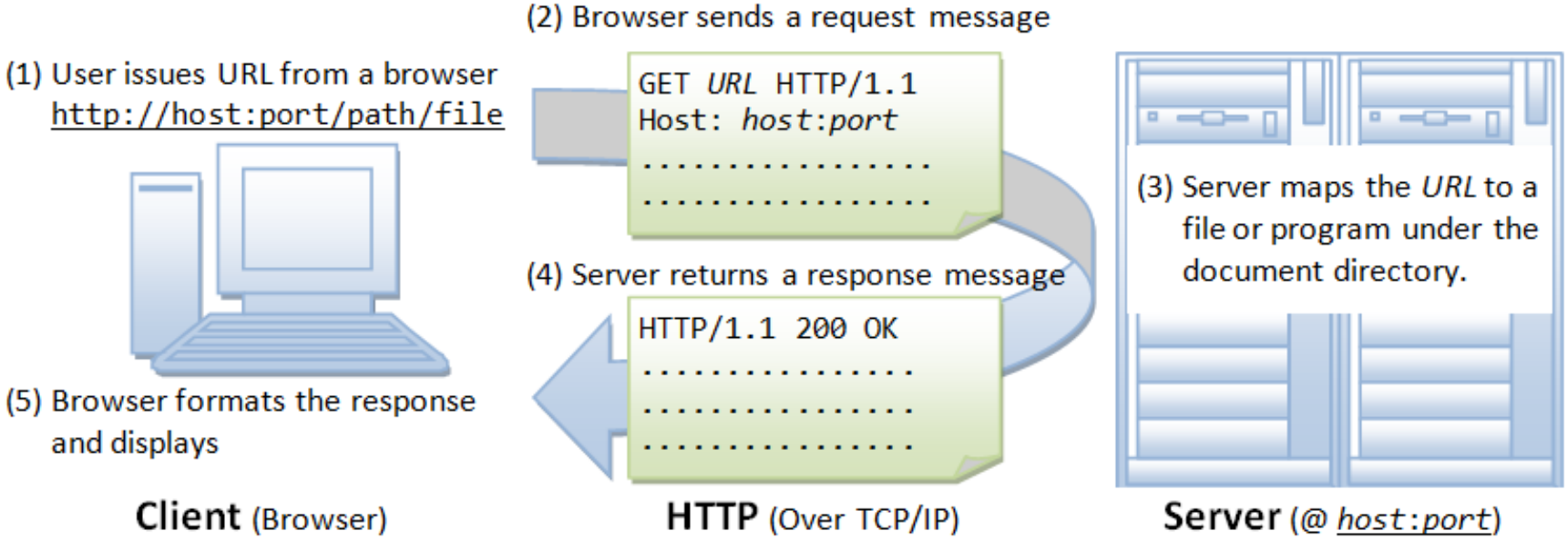
Exercise

Web Applications and Web Containers

- Web Application is a client–server computer program where:
  - The client (including the user interface and client-side logic) runs in a web browser
  - The server produces dynamic content (such as HTML pages) based on user actions
- Java Web Applications are managed and executed by special middleware called '**JavaEE container**' or '**Web / Servlet Container**'
- Web Container is a runtime environment for web application which handles:
  - Network connectivity
  - Lifecycle management
  - Application security
  - Concurrency
  - Transactions
  - Etc.



HTTP – HyperText Transfer Protocol



RESTful Service Implementation

- Java RESTful services are deployed within JavaEE or Web Container
  - Same as Web Applications
- Major differences between Web Application and RESTful Service implementations:
  - Data exchange format
    - HTML **vs.** JSON/XML/…
  - Frameworks used
    - SpringMVC/Struts2 **vs.** DropWizard/Restlet
      - Some frameworks, such as PlayFramework or **Spring Boot**, can be used for both
  - Specifications adhered to
    - ServletAPI **vs.** JAX-RS
      - Some implementations are 'specification agnostic', but follow common 'request dispatch' pattern
  - Client implementation
    - Browser **vs.** RESTful client (i.e., another application)

JSON

- **J**ava**S**cript **O**bject **N**otation (JSON)
  - A lightweight data-interchange format derived from the ECMAScript (JavaScript)
  - Syntax defined in ECMA-404 – The JSON Data Interchange Standard
  - Easy for humans to read and write, easy for machines to parse and generate
- JSON is built on two structures
  - Object (map): a collection of `name:value` pairs separated by comma
    - `{"key1":"value1", "key2":"value2"}`
  - Array (list): a collection of ordered `values` separated by comma
    - `["value1", "value2", "value3"}]`
- JSON `values` can be:
  - Strings (`"string1"`)
  - Numbers (`10, 3.141, 2.5E6`)
  - Boolean (`true` or `false`)
  - `null`
  - Another Object or Array (map of lists, list of maps, map of maps, list of lists)

JSON – Combining Objects and Arrays

- Objects and Arrays can be combined:
  - Family members aggregated by last name

```
{"Smiths": ["John", "Jane"],
 "Jones" : ["Ann", "Dave", "Rob"]}
```

  - List of individuals (with 'firstName' – optional)

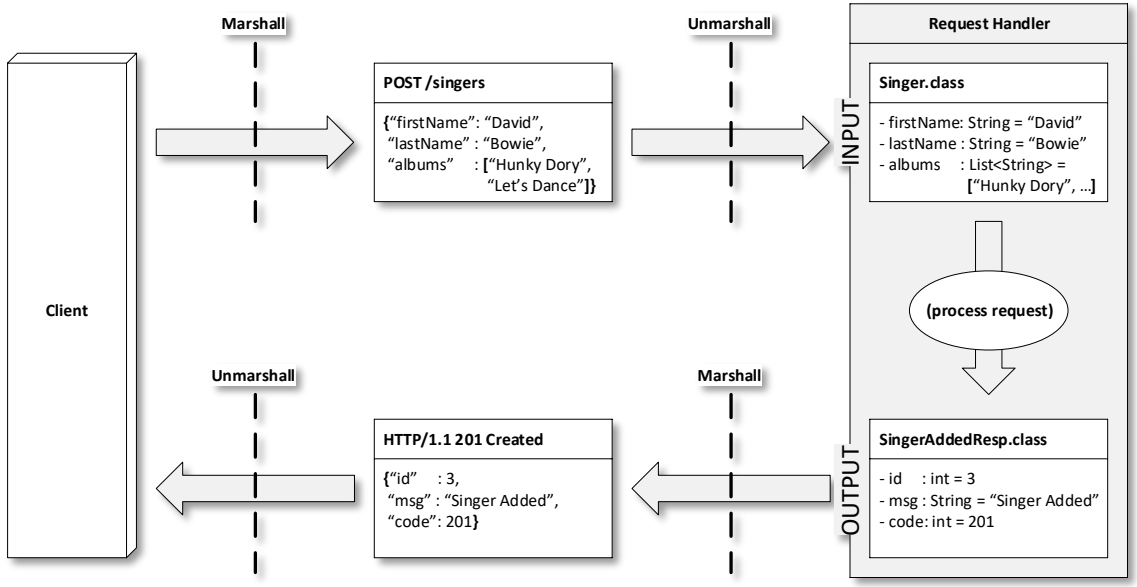
```
[
  {"lastName": "Doe"                          },
  {"lastName": "Smith", "firstName": "John"},
  {"lastName": "Smith", "firstName": "Jane"}
]
```

  - List of individuals (with 'firstName' – optional)

```
{"building1": {"1A": {"Smiths": ["John", "Jane"       ]},
                      {"Jones" : ["Ann", "Dave", "Rob"]}},
               {"2A": {"Kramer": ["Cosmo"             ]}}
}
```

Request/Response (De-)Serialization (from)to JSON

- Web Application is using JSP, JSF, or other templating engines to generate HTML response
- Spring Boot framework is using special libraries to automatically convert Java objects (POJOs – **P**lain **O**ld **J**ava **O**bjects) into JSON/XML and vice versa



Chapter Concepts

Web Services

HTTP and JSON

RESTful Services

Spring Boot

Exercise

REST Principles

- Application domain model (resources) are manipulated using standard set of actions
- Resources are identified by **U**niform **R**esource **I**dentifiers (URIs) and organized into collections in a tree-like structure
  - E.g.: http://mydealership.com/locations/***{locId}***/cars/***{carId}***
- Actions are normally represented via HTTP operations applied to **any** part of URI
  - `GET`, `POST`, `DELETE`, `PUT`, `PATCH`, etc.
- Data can be exchanged in various formats, though most common ones are **JSON** and XML
- Interactions are stateless
  - Actions are used to change the state of the resource one at a time
  - Each call is normally independent from each other
- Errors are handled via HTTP status codes
  - **200**: OK; **404**: Resource Not Found; **400**: Bad Request; **201**: New Resource Created

REST Operations – GET

- GET operation is a safe method and has no side effects ('R' in the CRUD)
  - Server-side content is unchanged

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


GET /inventory/cars        HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"   : "honda",
 "licPlate": "BDK032",
 "invId"   : 1}

HTTP/1.1 200 OK
[{"model"   : "honda",
  "licPlate": "BDK032",
  "invId"   : 1},
 {"model"   : "toyota",
  "licPlate": "GAV101"
  "invId"   : 2}]
```

REST Operations – POST

- POST operation is used to create resources ('C' in the CRUD)
  - Normal practice is to return a handler (id) to the created resource

Request

```
POST /inventory/cars      HTTP/1.1
Host: mydealership.com
{"model"   : "ford",
 "licPlate": "KYE903"}



GET /inventory/cars/3     HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 201 Created
{"model"   : "ford",
 "licPlate": "KYE903",
 "invId"   : 3}
                                  OR, simply,
{"invId"   : 3}


HTTP/1.1 200 OK
{"model"   : "ford",
 "licPlate": "KYE903",
 "invId"   : 3}
```

REST Operations – PUT

- PUT is an idempotent operation used to replace existing resource or create one if it doesn't exist ('C' and 'U' in the CRUD)
  - Resource is replaced as a 'whole'

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


PUT /inventory/cars/1      HTTP/1.1
Host: mydealership.com
{"model"    : "tesla",
 "licPlate": "AAA001"}

GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
   (with optional mirroring back of
    updated resource)


HTTP/1.1 200 OK
{"model"    : "tesla",
 "licPlate": "AAA001", … }
```

REST Operations – PATCH

- PATCH is an operation used to update existing resource ('U' in the CRUD)
  - Only some attributes of the resource are updated
  - Not used too often due to ambiguity of operation to be used (default is 'update')

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


PATCH /inventory/cars/1    HTTP/1.1
Host: mydealership.com
{"model":    "tesla"}


GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
   (with optional mirroring back of
    updated resource)


HTTP/1.1 200 OK
{"model"    : "tesla",
 "licPlate": "BDK032", … }
```

REST Operations – DELETE

- DELETE is an idempotent operation used to delete existing resource ('D' in the CRUD)

Request

```
GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com


DELETE /inventory/cars/1   HTTP/1.1
Host: mydealership.com



GET /inventory/cars/1      HTTP/1.1
Host: mydealership.com
```

Response

```
HTTP/1.1 200 OK
{"model"    : "honda",
 "licPlate": "BDK032", … }

HTTP/1.1 200 OK
   (with optional mirroring back of
    deleted resource)


HTTP/1.1 404 Not Found
```

HTTP Status Codes

- The HTTP protocol defines meaningful status codes
  - Which can be returned from a RESTful service
- Using status codes can help service consumers
  - Determine how to understand the service response
  - Especially when errors occur
- What is status code 418?

https://en.wikipedia.org/wiki/List_of_HTTP_status_codes

HTTP Status Codes (continued)

200 OK	Response to a successful request
201 Created	Response to POST that results in a resource creation
204 No Content	Response to a successful request that does not return a body
400 Bad Request	The request was malformed
401 Unauthorized	Either invalid or missing authentication details in request
403 Forbidden	User does not have access to the requested resource
404 Not Found	We've all been here before
405 Method Not Allowed	The HTTP method is not allowed for this user
410 Gone	The resource is no longer available
418 ?	

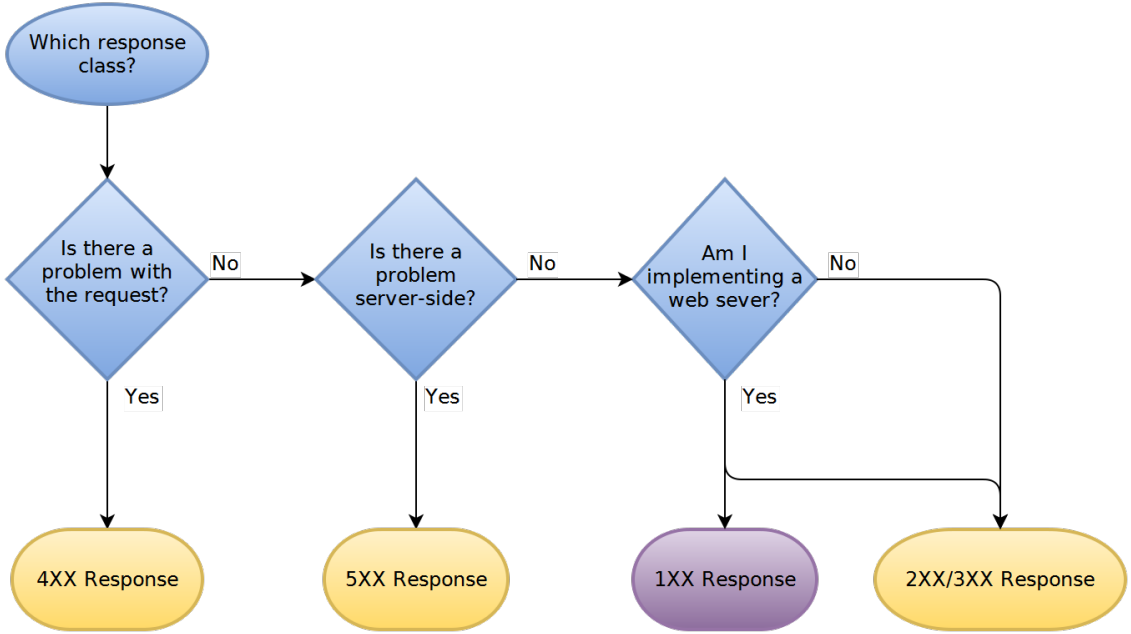
Why Should I Use a Status Code?

- They communicate to the RESTful client
  - When an exceptional event occurs
  - When some special behavior is required
- Many status codes represent situations that are worth handling with a special response
- Many widely used APIs are using them
  - A convention is being created
  - Following that convention makes it easier for users of your RESTful service
  - https://gist.github.com/vkostyukov/32c84c0c01789425c29a

What Status Should I Return?

- The following flowcharts answer this question
  - From http://racksburg.com/choosing-an-http-status-code/

- The flowcharts for each category of response are too big to fit on these slides
- Visit the above URL to see them



How to Return a Status Code

- Two main methods for returning an HTTP status code response
- Return a Response object
  - That wraps your Java return object
  - And adds the status code
- Throw a WebApplicationException
  - Will turn this into a Response object
  - And send it back to the client with the status code

RESTful API: Best Practices

- Use correct HTTP method names

Resource	GET read	POST create	PUT update	DELETE delete
`/cars`	Returns a list of cars	Create a new car	Bulk update of cars	Delete all cars
`/cars/711`	Returns a specific car	Method not allowed (405)	Updates a specific car	Deletes a specific car

- Use nouns, not verbs, in the URI
  - That is, do **NOT** use `/addCar`, `/deleteCar`, `/updateCar`
  - Whenever 'special' actions are to be communicated, append 'verb' to the resource name:
    - `POST /accounts:transferMoney`
    - `{"fromAccount": "0123", "toAccount": "4567", "amount": 100}`
  - In most cases, special verbs can be avoided, though might require thinking out-of-box
    - `POST /transfers`
    - `{"fromAccount": "0123", "toAccount": "4567", "amount": 100}`

RESTful API: Best Practices (continued)

- To implement concurrency and pagination use '**ETag**' together with '**If-Match**'
  - Server sets **ETag** HTTP response header based on the content on the response
  - Client mirrors back ETag value in the **If-Match** HTTP request header together with request
  - Server processes the request only if recalculated value of ETag for the resource matches the value of **If-Match** request header
    - Protects against concurrent modification of resource (or collection during iteration)
    - Facilitates 'Read-Modify-Write' pattern
    - Facilitates 'continuation' reads
- Use URI query parameters for filtering, sorting, field selection, pagination
  - `GET /cars?color=red&seats=2`
    `&sort=manufacturer,model`
    `&fields=manufacturer,model,id,color`
    `&offset=10&limit=5`

RESTful API: Best Practices (continued)

- Version your API to avoid breaking existing clients when API changes
  - http://mydealership.com/api/**v1**/inventory/cars/1
    - Use a simple ordinal number
    - Avoid dot notation such as 2.5
- Use correct HTTP status codes to communicate both success and failures
  - See https://tools.ietf.org/html/rfc7231 for details; keep in mind that industry practice might deviate occasionally
  - Duplicate HTTP status code in the body of the Response message:

```
HTTP/1.1 404 Not Found
{"Message": "Not Found",
 "Code"   : 404}

HTTP/1.1 201 Created
{"Message": "Created",
 "Code"   : 201}
```

```
HTTP/1.1 200 OK
{"Response" : {"model"   : "honda",
               "licPlate": "BDK032",
               "invId"   : 1},
 "Message"  : "Ok",
 "Code"     : 200}
```

RESTful API: Best Practices (continued)

- Build the API with consumers in mind
  - Make sure hierarchy is easy to navigate for your target clients/application domain
  - Add filtering, sorting, pagination capabilities
- Create two endpoints per resource
  - The resource collection (e.g., `/cars`)
  - Individual resource within the collection (e.g., `/cars/{carId}`)
- Alternate resource names with IDs as URL nodes where needed

```
/LEVEL 1                 /LEVEL 2                    /LEVEL 3   / …
-------------------------------------------------------------------
/locations/{locId}       /cars          /{carId}
                         /staff         /{empId}
                         /sales         /{yyyymmdd}
/employees/{empId}
/accounts /{accountId}/transactions/{txnId}
```

Richardson Maturity Model

- Dr. Leonard Richardson developed a model that breaks down the principal elements of a REST approach into three steps
  - http://martinfowler.com/articles/richardsonMaturityModel.html
- Model defines four maturity levels of RESTful API
  - Level 0:
    - RPC-style API, usually with a single endpoint
  - Level 1 – Resources:
    - Resources are introduced; multiple endpoints based on the structured URI
  - Level 2 – HTTP Verbs:
    - Same as Level 1 + HTTP verbs to distinguish between operations
  - Level 3 – Hypermedia Controls:
    - HATEOS (Hypertext As The Engine Of Application State) – 'Discoverable' API
    - Response message contains **WHAT** we can do next and **HOW** to do it
      - Think hyperlinks on HTML pages

Exercise: Create URI Resource Hierarchy

- Pick a subject domain
  - Can be anything: HR system, car dealership, inventory system, etc.
- Create a hierarchy of resources following best practices
- CHALLENGE!
  - Is there any other way to navigate your subject domain?

Chapter Concepts

Web Services

HTTP and JSON

RESTful Services

Spring Boot

Exercise

Spring Boot

- Makes it easy to create standalone applications
  - Very little configuration required
  - Spring and third-party libraries included
- Some of the key features include:
  - Applications begin with main method
  - Embed Tomcat, Jetty, or Undertow directly in application
  - Starter POMs provided simplify Maven configuration
  - Automatically configures Spring whenever possible
  - Provides production-ready metrics, health checks, and externalized configuration
- An 'accelerator' to build applications fast
  - Spring MVC, Spring Security, and other Spring libraries can be used WITHOUT Spring Boot

Traditional JavaEE Frameworks vs. Spring Boot

Traditional frameworks

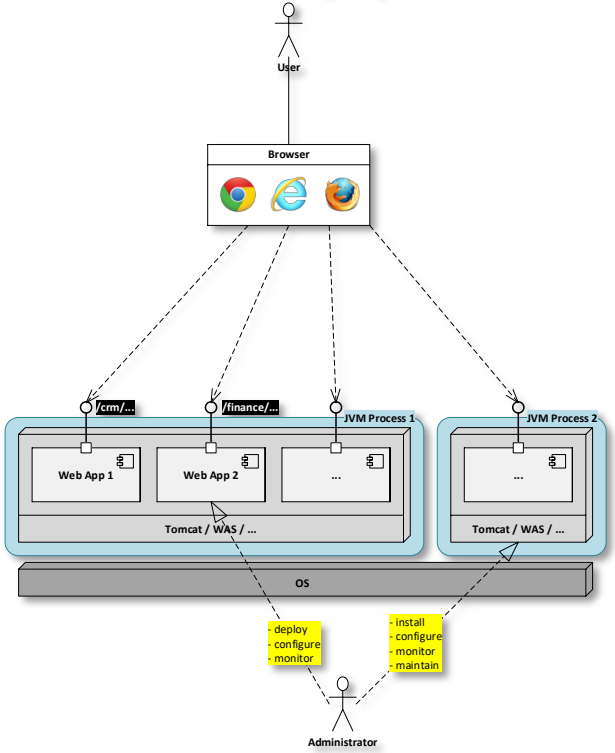
- Pick favorite MVC framework
- Download additional libraries
  - Make sure to use the right version
  - Add Spring framework if needed
- Compile and create WAR file
- Install and configure application server
- Deploy WAR file to application server

Spring Boot

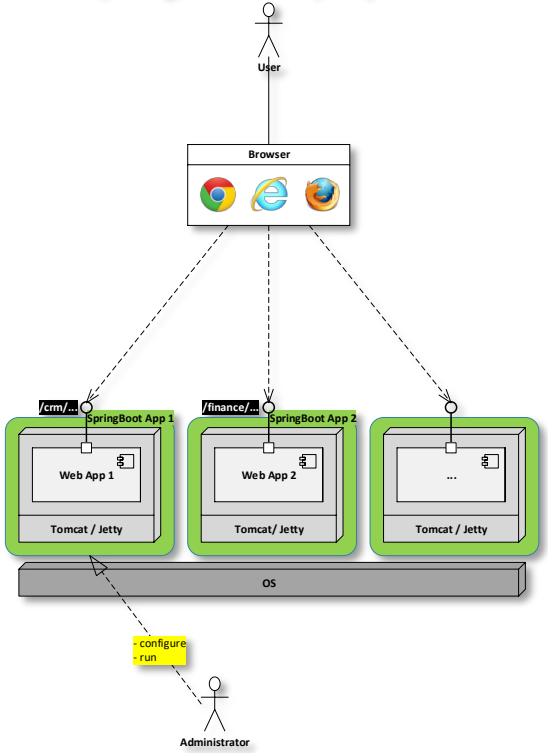
- Put Spring Boot library in project dependencies
- Implement web application to conventions of Spring Boot
- Compile & run!

Traditional vs. Spring Boot Deployment

Traditional Deployment



Spring Boot Deployment



Hello World with Spring Boot

- Spring Boot provides a parent POM and also starter projects
  - Have dependencies required for application type
    - For example, starter Web has dependencies for Spring MVC and REST applications

```
<parent>
	<groupId>org.springframework.boot</groupId>
	<artifactId>spring-boot-starter-parent</artifactId>
	<version>1.5.4.RELEASE</version>
</parent>

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

A Simple Service

- The service will return the string "Hello World!" when requested

```
@SpringBootApplication
public class HelloApplication {

    public static void main(String[] args) throws Exception {
        SpringApplication.run(HelloApplication.class, args);
    }
}
```

Entry point of application

```
@RestController
public class HelloService {

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

`/hello` routed to this method

A Simple Service Explained

- `@RestController` indicates class represents one or more endpoints
- `@RequestMapping` defines routing information for the services
- `@SpringBootApplication` tells Spring to detect dependencies
  - Configure application based on these dependencies
  - Equivalent to using `@Configuration`, `@EnableAutoConfiguration`, and `@ComponentScan` with their default attributes
- The `main` method
  - Delegates work to `SpringApplication`
    - Bootstraps application starting Tomcat
- *Gotcha:* make sure `@SpringBootApplication` bean is located in the package at the 'top'/'above' other annotated beans

Running a Spring Boot Application

- The application can be started using a Maven run goal
  - Provided by the starter parent POM
  - Can choose port number Tomcat starts on
    - Default port is 8080
- `mvn -Dserver.port=9090 spring-boot:run`

A Currency Service

- Following examples show a service returning currency data
  - Data will be serialized into JSON:
    - `["USD", "CAD", "GBP"]`

```
@RestController
public class CurrencyService {
  private final Logger log = LoggerFactory.getLogger(this.getClass());

  @RequestMapping(value="/currencies", method = RequestMethod.GET)
  public List<Currency> getCurrencies(){
    return Arrays.asList(Currency.values());
  }
}
```

Accessed by `/currencies` and `HTTP GET` only

Receiving Client Data

- JSON data sent from client will be marshalled to Java Objects
  - `@RequestBody` indicates data posted from client
  - Unmarshalling happens automatically

```
{
 "currency": "EUR",
 "amount"  : 11,
 "side"    : "BUY"
}
```

```
public class MarketOrder {
  private Currency currency;
  private int      amount;
  private Side     side;
}
```

```
@RestController
public class OrderService {
  private final Logger log = LoggerFactory.getLogger(this.getClass());

  @RequestMapping(value="/order", method = RequestMethod.POST)
  public void addOrder(@RequestBody MarketOrder order){
    // process order
    log.info("Order received "+ order);
  }
}
```

POJO with properties matching JSON property names

Processing Request Parameters and Path Variables

- `@RequestParam("<param name>")`
- `@PathVariable <named uri segment>`
- Example:

`/cars/711?fields=model`

```
...
  @RequestMapping(value="/cars/{carId}", method = RequestMethod.GET)
  public Car getCarDetails(@PathVariable("carId")  int    carId,
                           @RequestParam("fields") String fieldsToReturn){
    // return car model
    log.info("Car details ("+ fieldsToReturn +") returned");
    return carRepository.findCar(carId);
  }
...
```

`RequestMapping` Annotation Shortcuts

- 'Verb-specific' specializations of `RequestMapping` annotation (SpringBoot v1.4+)
  - `@GetMapping`
  - `@PostMapping`
  - `@PutMapping`
  - `@DeleteMapping`
  - `@PatchMapping`
- Most attributes can be applied both at class (`@RestController`) and method levels
  - `GET /inventory/cars/711`

```
@RestController
@RequestMapping("/inventory")
public class CarInventoryService {

  @GetMapping("/cars/{carId}")
  public Car getCarDetails(@PathVariable int carId){...}
}
```

Content Negotiation

- The REST controllers can accept and respond with data in different formats
  - Controller inspects 'Content-Type' and 'Accept' headers set by the client and decides whether it can process the request in 'Content-Type' format and respond in the format indicated by 'Accept' header:

Request (Content-Type = application/json,
Accept = application/xml,
application/json)

```
POST /cars                    HTTP/1.1
Host          : mydealership.com
Content-Type: application/json
Accept        : application/xml,
                application/json
{"model"    : "ford",
 "licPlate": "KYE903"}
```

Controller (consumes = application/json,
produces = application/xml)

```
HTTP/1.1 201 Created
Content-Type: application/xml
<invId>3</invId>
```

Controller (consumes = application/json,
produces = application/json)

```
HTTP/1.1 201 Created
Content-Type: application/json
{"invId": 3}
```

Content Negotiation (continued)

- Controller capabilities are defined via 'produces' and 'consumes' attributes of `@RequestMapping` annotation

```
@RequestMapping(value="/cars", method = RequestMethod.POST,
        produces={MediaType.APPLICATION_JSON_VALUE, MediaType.TEXT_XML_VALUE},
        consumes=MediaType.APPLICATION_JSON_VALUE)
public Car createCarRecord(){
  return new Car(…);
}
```

Produces JSON (default) or XML

Consumes JSON

- To serialize data into XML, add the following dependency to pom:

```
<dependency>
        <groupId>com.fasterxml.jackson.dataformat</groupId>
        <artifactId>jackson-dataformat-xml</artifactId>
</dependency>
```

Exception Handling

- Any unhandled exception causes the server to return an HTTP 500 response

```
{ "timestamp": 1516773431477,
  "status"   : 500,
  "error"    : "Internal Server Error",
  "exception": "com.artilekt.bank.business.AccountNotFoundException",
  "message"  : "Account [eebb2ced] not found",
  "path"     : "/accounts/eebb2ced" }
```

- There are two ways to customize exception handling
  - Per exception
    - By annotating custom exceptions with `@ResponseStatus` annotation
  - Globally
    - By creating classes annotated with `@ControllerAdvice` annotation

Exception Handling Customization – Per Exception

- Annotate custom exceptions with `@ResponseStatus` and define HTTP error code

```
@ResponseStatus(HttpStatus.NOT_FOUND)
public class AccountNotFoundException extends RuntimeException {
    ...
}
```

- Exceptions thrown from within your code …

```
public Account findAccountByNumber(String accountNumber) {
    Account acc = dao.getAccount(accountNumber);
    if (acc == null)
        throw new AccountNotFoundException("Account ["+accountNumber+"] not found");
    return acc;
}
```

- … would be automatically converted to JSON

```
{ "timestamp": 1516940765026,   "status": 404,   "error": "Not Found",
  "exception": "com.artilekt.bank.business.AccountNotFoundException",
  "message"  : "Account [eebb2ced] not found",   "path": "/accounts/eebb2ced" }
```

Exception Handling Customization – Global

- To fully customize error response, define `@ControllerAdvice` class(es)

```
@ControllerAdvice
public class ClientExceptionHandler extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ ClientDuplicateException.class })
    protected ResponseEntity<GenericErrorResponse> handleConflict(RuntimeException ex,
                                                                   WebRequest request) {
        GenericErrorResponse response = new GenericErrorResponse();
        response.setErrorCode("Client Duplicate");
        response.setErrorMessage(ex.getMessage());

        return new ResponseEntity<>(response, HttpStatus.CONFLICT);
    }
}
```

```
public class GenericErrorResponse {
    private String errorCode;
    private String errorMessage;
}
```

```
{
  "errorCode"    : "Client Duplicate",
  "errorMessage": "Client [Client [id = a001]]
                    already exists" }
```

Spring Boot Actuator

- Includes a number of features that let you monitor and manage your application
- Endpoints are made available over HTTP; for example:
  - `/actuator/beans` – lists all Spring beans in the application
  - `/actuator/configprops` – list of all `@ConfigurationProperties`
  - `/actuator/metrics` – list of metrics for the application
  - `/actuator/health` – application health information
  - Many more
- Enabled by including the following dependency:

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

Spring Boot DevTools

- Spring Boot DevTools improves the development-time experience when working on Spring Boot applications
  - **Automatic Restart** of application whenever files on the classpath change
  - **Live Reload** triggers a browser refresh when a resource is changed
    - Requires browser plugin
  - **Global Settings** properties defined in `~/.spring-boot-devtools.properties` file which will apply to *all* Spring Boot applications on your machine that use devtools
  - **Remote Applications** enable 'live' deployment of updates to remote server as well as remote debugging
  - **H2 Web Console** to view content of in-memory H2 database, available at `/h2-console`
    - http://www.h2database.com/html/quickstart.html#h2_console

Spring Boot DevTools (continued)

- To include devtools support, simply add the module dependency to your build

```
<dependencies>
   <dependency>
      <groupId>org.springframework.boot</groupId>
      <artifactId>spring-boot-devtools</artifactId>
      <optional>true</optional>
   </dependency>
</dependencies>
```

Testing Services with Postman

- Postman is a tool that can be used to test REST services
  - Enables messages to be configured
  - Service responses to be viewed
  - Allows to create 'collections' of requests, similar to 'SOAP UI' test suites
    - Use this to 'replay' messages during service development
- Your instructor will now demonstrate Postman
- RestAssured is a library for writing tests for REST services
  - Provides DSL that supports *given-when-then* structure
  - Details found at `rest-assured.io`

Spring Boot Details

- Full details of Spring Boot can be found at:
  - http://docs.spring.io/spring-boot/docs/current/reference/
- Skeleton Spring Boot project generator:
  - http://start.spring.io/

Chapter Concepts

Web Services

HTTP and JSON

RESTful Services

Spring Boot

Summary

Chapter Summary

In this chapter, we have:

- Introduced REST Web Services
- Introduced Spring Boot
- Learned how to write REST Web services using Spring Boot