Spring 5 : Beginner to Guru

With Notes from CouraRA Spring Specialization

What is Spring

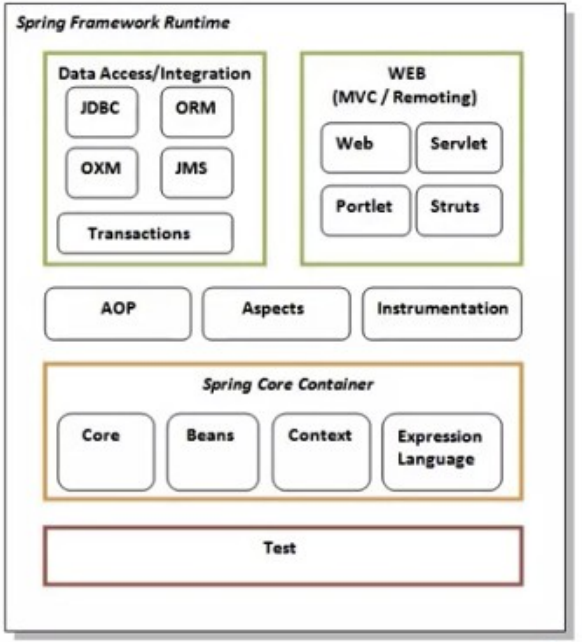
- provide a configuration model for Java-Based Enterprise Application
  - Spring focus on the plumbing of applications so teams an focus on application business logic
  - Spring has support for each tier of the network
- Spring is organize in an modular fashion
- Spring has a framework makes it testable
- Promotes decoupling and reusability
- Removes common codes issues like leaking connection and more
- Build on design patterns
- Spring is both a
  - container
    - Creates objects and makes them available to your application
    - Use a declarive XML Strucutre to define components that live in the container
  - Frameworks
    - Ex. templates provide abstractions

What is Spring

Core Modules

What does Spring Cover

- The Spring Framework comprises of many modules such as core, beans, context, expression language, AOP, Aspects, Instrumentation, JDBC, ORM, OXM, JMS, Transaction, Web, Servlet, Struts etc.
- These modules are grouped into;
  - Test
  - Core Container
  - Aspect Oriented Programming (AOP)
  - Aspects
  - Instrumentation
  - Data Access / Integration,
  - Web (MVC / Remoting)
- The Spring Core container contains core, beans, context and expression language (EL) modules



Spring Initalizr
Open Project In Intellij

- A website to download a preconfigured project to our specifications
  - start.spring.io
  - For this project we will use JPA, H2, Thymeleaf, Web, Actuator
  - Provides the developer with a zip file
- Open Project in Intellij
  - Spring Boot use parent boot which provides a set of curated dependencies
    - Inheriting all the version information from them only get the items that you select in spring.io in the POM file
    - Not including hibernate , but it will be a dependency of spring-boot-start-data-jpa
    - Thymeleaf will get automatically configured by SpringBoot
  - At the bottom of intellij there is a terminal icon which will create a terminal at the current selected directory
  - Spring Boot has a maven wrapper
    - Use ./mvnw spring-boot:run spring-boot is a maven goal and run is an attribute

Using JPA Entries

- What is JPA
  - JPA is a specification not a concrete implementation
  - ORM → Object Relational Mapping
  - JPA offers Java Developers database Independence
    - One API will support many relational database
- Convert POJO into JPA Classes
  - Start with two POJO Book and Author
  - Javax.Persistence is the official JPA Package
  - Above each POJO put @Entity
  - An example of leakage
    - In each POJO we have a long id; to uniquely identify the POJO, but In the OO World we do not need it , but in the DB world we do.
    - Example
      - @Id
      - @GeneratedValue(strategy = GenerationType.AUTO)
      - Private long getId();
  -

Using JPA Entries

- In the Author class we have a set of books which should have the relationship many Authors to many books
  - @ManyToMany
  - Private Set<Book> = new HashSet<>()
- Spring Boot
  - In resources /application.properties there is a file to put properties
    - spring.h2.console.enabled = true
- With minimal information JPA gave use Author, Author_Books, Book_Authors and Book we only want Authors, Books
  - Problem : Hibernate is trying to define both way of the direction
  - In the Author POJO
    - @ManyToMany(mappedBy=”authors”)
      - Create a relationship with authors being the target side
  - In the Books
    - @ManyToMany
    - @JoinTable(name=”another_book”, joinColumn = “book_id”), inverseColumns = @JoinColumn(name=”author_id”)
      - // Creates a singular id value
    - private Set<Author> authors = new HashSet<>();
    - For Bidirectional : To the Publishers add @ManyToMany(mappedBy=”publishers”)privateSet<Book> = new HashSet<>();
  - Now we get a singular table (AUTHOR_BOOK) to define the many to many relationship
  - Hibernate will generate the schema DDL based on our mappings.

Equality in Hibernate
Spring Data JPA Repositories

- Hibernate recommends when working with sets to implement hashcode and equals
- Spring Data JPA Repositories
  - Provides an implementation of the Repository Pattern
    - Has methods for retrieving domain object should delegate to a specialized Repository object such that alternative storage implementations may be interchanged
    - Easily substitute the persistence layer
      - Ex. going form SQL to No SQL
  - Spring Data JPA
    - User Hibernate for persistence to supported RDBS systems (just about any major relational database)
    - You extend Java Repository Interface
    - Provides implementations at run time
    - No SQL Required, No DAO, No Transaction Management, but managed by the framework
  - Spring Data JPA is going extend the interface and give our entity type and inject the implementation of the DAO
  - Example
    - Make a new package for repositories
    - Public interface AuthorRepository extends CrudRepsotiry<Author, Long> {} // Does all CRUD

Publisher Relationships

- Admittedly I JPA mapping first so there will not be much here.
- @JoinColumn(name=”publisher_id”)) // Add a publisher id to the Book Record to track the publisher to create a foreign key relationship
- Hibernate generates the SQL

H2 Database Console

- Run the Spring Boot Web Application
- In the console that tomcat has started on Port 8080
- WhiteLabel Error Page is the SpringBoot Error Page
- In the Spring Boot Application Properties has several h2 properties
  - Set the spring.h2.console.enabled = true to enable the database
- From SpringBoot you can get the dataurl which will show your tables not SpringBoot

Initializing Data with Spring Framework Events

- When the ContextRefreshEvent gets thrown the code will be executed.
- ApplicationListener
- Example
  - @Component // Make this into a spring bean
  - Public class DevBootStrap implements ApplicationListener<ContextRefreshEvent> { // Application Listener listening for ContextRefreshEvent
    - Private void initData() {
      - Author eric = new Author(“Eric”, “Evans”);
      - Book ddd = new Book(title: “Domain Driven Design, isbn: “1234”, publisher: “Harper Collins”);
      - eric.getBooks().add(ddd); ddd.getAuthors().add(“eric”);
      - authorRepository.save(eric);
      - bookRepository.save(ddd); }
    - @Override
    - Public void onApplicationEvent(ContextRefreshEvent contextRefreshEvent) {InitData(); }
    - private AuthorRepository authorRepository;
    - Private BookRepository bookRepository;
    -
    - // Dependency Injection will happen here
    - Public DevBootStrap(AuthorRepository authorRepository, BookRepository bookRepository) {
      - This.authorRespository = authorRepository; this.bookRepository – bookRepository; }
  - }

Introduction to Spring MVC

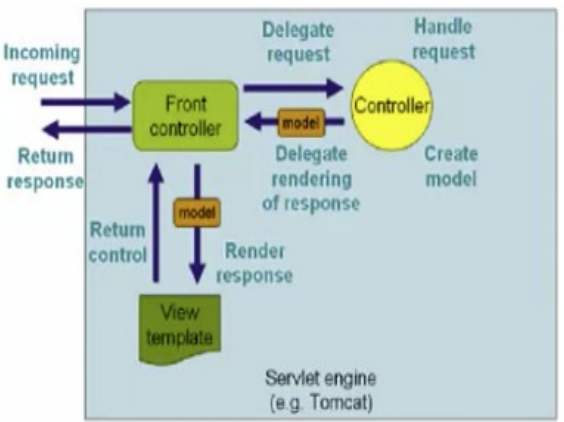
- In MVC the controller determines how to get the model and return it to the view.
  - Does a great job at search concerns model, state, data business logi
  - View render the component for the client
  - Controller → A light traffic cop
    - Return the model (POJO) to the view component
    - interface between view and model intercepts all request from the view and model
  - Thymeleaf → Takes the model and with the template creates a html file
- Goals of Spring MVC
  - Simplify the task of creating web applications
  - Encourage good architecture
  - Integrate easily with core modules

Introduction to Spring MVC

- In MVC the controller determines how to get the model and return it to the view.
  - Does a great job at search concerns model, state, data business logi
  - View render the component for the client
  - Controller → A light traffic cop
    - Return the model (POJO) to the view component
    - interface between view and model intercepts all request from the view and model
  - Thymeleaf → Takes the model and with the template creates a html file
- Goals of Spring MVC
  - Simplify the task of creating web applications
  - Encourage good architecture
  - Integrate easily with core modules

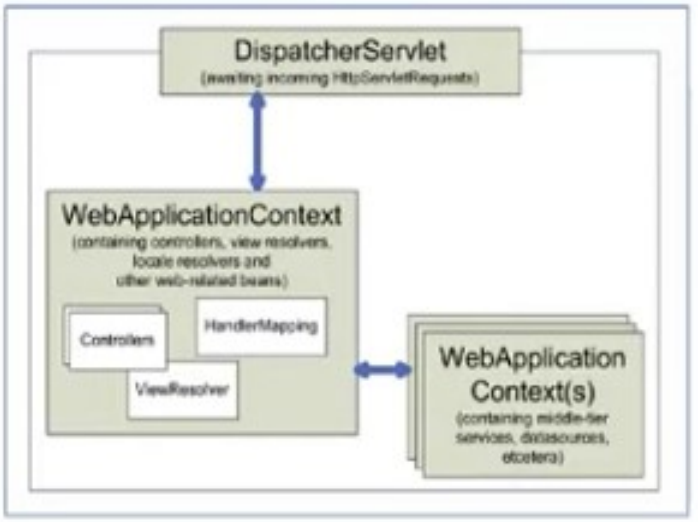
Introduction to Spring MVC

- Spring MVC
  - Wildcards URLs go through a single controller that delegates to sub controllers to get a the core application
    - Response from the subcontroller to the FrontController informs the frontController where to dispatch next
  - Client sends a request
    - The request goes to the dispatcher servlet (Front Controller) which determines the controller
      - Follows the FrontController pattern providing a single point of entry into the application
        - Through configuration it will delegate to subcontrollers which will interact with your application business logic
        - The Controller will return data that the view module to the front controller and send the data received to the view for display
    - Handler Mapping
      - Convert the context path in a controller method
      - Controller (Controller Method)
  - Service – Gets the data instead of putting the code in the controller
    - Data is then returned to the dispatcher servlet
  - View
    - The dispatcher servlet calls a function that knows how to handle the view with the model
    - The model will get passed to an templating engine and the page will be create the page

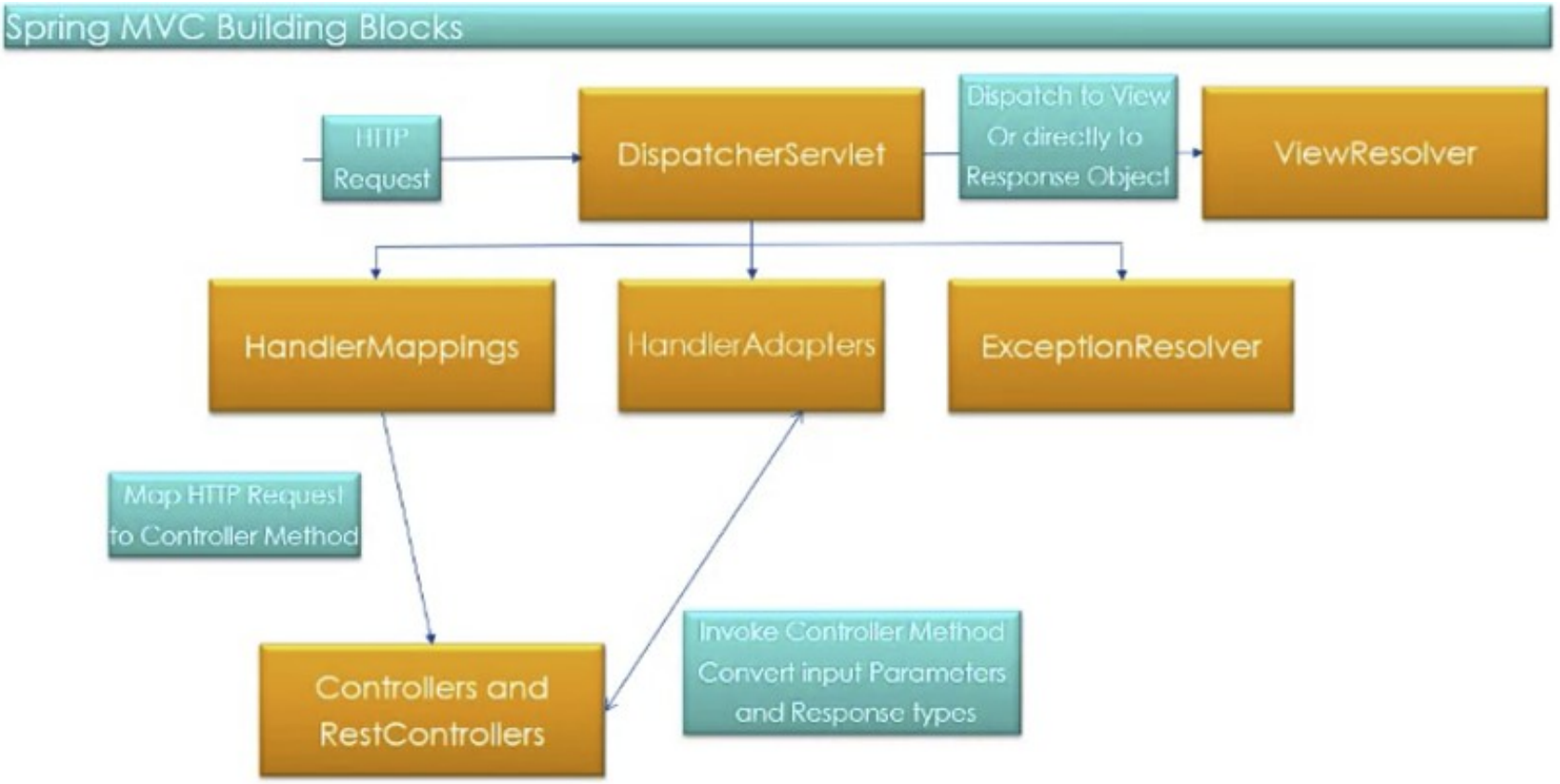


Introduction to Spring MVC

- Each Dispatcher Servlet has its own WebApplicationsContext when the servlet is initialize.
  - Creating the dispatcher and delegate to rest controllers and delegate to to the second WebApplication call Root Context
- Second WebApplication called the Root Context – know we components for the DispatcherServlet. They can communicate with each other.
  - Concerned with beans
  - Non Web stuff



Introduction to Spring MVC



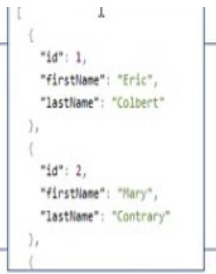
Introduction to Spring MVC

- Spring MVC Building Blocks
  - An Http Request enters the Dispatcher Servlet
    - Dispatcher Servlet is configured through metadata in Spring to get to the controller or rest controller
    - Achieves through a cross reference where this URL got the method of the controller or rest controller
      - Called the HandlerMappings
    - The Http Request get converted into Java Object which can be passed to the controllers
      - The handler adapters convert it from Http Request to Java Objects to Http Response
    - When the rest controller/controller provides a response to the dispatcher servlet
      - Either a view will be selected or it will write directly to the response object
    - if anything goes wrong we have ane exception resolver to generate some error and response accordingly to the requestor
- HandlerMappings – Deliver a HandlerExecutionChain by mapping an Http Request to a controller method then the dispatcher servlet will execute the handler
- HandlerAdapter – use by the Dispatcher Servlet to invoke the controller method. Has knowledge of XML/JSON
- Exception Resolver – Resolves uncaught exceptions
- View Resolver – Either dispatching ot a view or writing directly to the containers response object
- Controller/Rest Controller

Rest Controller

- Request and response are built around the transfer of representations of resources
  - Don’t send the document, but a representation of that document
  - Capture the current or intended state of a resource
- Can get arguments : HttpServletRequest request
  - request.getParameter(“employee”)
- The response type from a controller can include java type. Below we return a Collection of data to verify that the WebApplicationContext of the DispatcherServlet is talking to the root context of our own beans
  - Can send employee objects back, but need to convert them
  - The Browser get json back
-

```
@Inject  private EmployeeDao dao;
@GetMapping(produces=MediaType.APPLICATION_JSON_VALUE)
public Collection<Employee> getAll() {
    return dao.getAll();
}
```




```
@RestController
@RequestMapping("/")
public class HomeController {
    @GetMapping
    public String home() {
       return “hello";
    }
    @GetMapping(path=“/abcd”)
    public String getAbcd() {
       return “World";
    }
}
```

Above our URL’s are to Home Controller HTTP GETS
“/” And “/abcd”

Configuring Spring Controllers

- Associate a controller method with a request path
- Annotate Controller Class with @Controller
  - Register the class as a Spring Bean and as a Controller in Spring MVC
  - To Map methods to http request paths use @RequestMapping
- Best Practice: create a new package for the controller
- Example }
  - @Controller
  - Public class BookController {
    - private BookRepository bookRepository;
    - Public BookController(BookRepository bookRepository) {
      - this.bookRepository = bookRepository; }
    - @RequestMapping(“/books”)
    - Public String getBooks(Model model) { // Spring MVC passes the model implementation when the controller method is invoked at runtime
      - model.addAttribute(“books”, bookRepository.findAll()): // Get a list of book from the db and add an attribute to the model
      - return “books” // Return a view name } // books is the Thymeleaf view
  - }

Spring MVC using Request Parameters

- Data in URL
  - Use URI to link to named resources
  - Could provide the template for http://example.org/foo/{userId}
  - Implementation
    - As a Spring Managed Bean our Controller can have other Spring managed beans injected into it via package scanning
    - @RequestMapping(“/employee”)
    - @RestController
    - public class EmployeeController {
      - @Inject private EmployeeDao dao;
      - @GetMapping(“{id}”)
      - public Employee get(@PathVariable(“id”) long id) { return dao.getOne(id); } }
    - To access the above http://localhost:8081/work/employee3
    - The response from the method is transformed into json which is an inject ResponseEntity Object that contains header and payload
- Example of a custom response entity
  - @GetMapping(“{id}”)
  - public ResponsoeEntity<Employee> get(@PathVariable(“id”) long id) {
    - return RepsonseEntity.ok().header(“greeting”, “Hello Word).body(dao.getOne(id));

Spring MVC using Request Parameters

- ResponseEntity.badRequest().build();
  - With no body need the build command
- Using the @RequestParam we can get data even if there is no query string
  - A RequestParameter can be Optional , but a PathVariable get data directly from the url and cannot be optional
    - URL with PlacesHolders are used by the PathVariable
  - @GetMapping(path="/single")
  - public Employee get(@RequestParam(name="id", defaultValue="4", required = false) long id {}
    - parse employee/single?id=4
  - ------
  - @GetMapping(path="/single") // Modern way of doing it. Use when data can be optional
  - public Employee get (@RequestParam(name="id") Optional<Long> optional) { Long id = optional.orElse(5L) }
    - parse employee/single
- We can get at the request headers
- 1st Way – No longer used. Injecting the whole request
- 2nd Way – If you don't have the parameter it become a problem
- 3rd Way Best

We can also get at the Request Headers

```
@RequestMapping("/myHeader")
@RestController
public class HeaderController {

    @GetMapping
    public String get(HttpServletRequest request) {
        return "hello " + request.getHeader("user-agent");
    }
    @GetMapping("/mandatory")
    public String getHeader(@RequestHeader("user-agent") String agent) {
        return "hello " + agent;
    }
    @GetMapping("/optional")
    public String getHeader(@RequestHeader("user-agent") Optional<String> optional) {
        return "hello " + optional.orElse("no agent");
    }
}
```

http://localhost:8080/myHeader

http://localhost:8080/myHeader/mandatory

http://localhost:8080/myHeader/optional

Spring MVC using Request Parameters

- Returning collections is easy
- It will return a json array

```
@GetMapping
public Collection<Employee> getAll() {
    return dao.getAll();
}
```

- However, we can build on this by providing mandatory (@PathVariable) parameters and optional , @RequestParams

```
@GetMapping("/search/{nme}")
public Collection<Employee> getEmployeesFiltered(@PathVariable("nme") String name,
@RequestParam("lastName") Optional<String> optional) {
return dao.getAll().stream()
    .filter(p-> p.getFirstName().equalsIgnoreCase("Eric"))
    .filter(p-> p.getLastName().contains(optional.orElse("")))
    .collect(Collectors.toList());
}
```

```
[
 {
  "id": 1,
  "firstName": "Eric",
  "lastName": "Colbert"
 },
 {
  "id": 4,
  "firstName": "Eric",
  "lastName": "Stewart"
 }
]
```

- http://localhost:8081/work/employee/search/Eric?lastName=o
- http://localhost:8081/work/employee/search/Eric

Spring MVC using Request Parameters

- If used @PathVariable can be made Optional — True / False
- If used @RequestHeader is can be made Optional — True / False
- If used @RequestParam is always Mandatory — True / False
- @RequestBody is always Mandatory for receiving a Request Payload — True / False
- The “path” attribute of @GetMapping can be left out — True / False
- The “produces” attribute of @GetMapping is mandatory — True / False

Spring MVC – Rest Post

The Response Entity contains the URI to retrieve the data

- Example
  - @PostMapping
  - public ResponseEntity<Receipt> add(@RequestBody Employee employee) {
    - dao.add(employee)
    - Receipt receipt = new Receipt();
    - receipt.setDate(new Date().toString());
    - receipt.setMessage("Employee added to department " + employee.getDeptId());
    - return (employee != null) ? ResponseEntity.accepted().location(URI.create("/" + employee.getId()) : ResponseEntity.badRequest.build();
    - }
- @Request body – A method parameter should be bound to the body of the web request. The body is converted to Java
- The payload will be returned as XML
- When we send a textual pay to a service it passes through the Dispatcher Servlet, the HttpAdapters and inject into the controller method
  - @postMapping(– Get the information from the accept header. If no accept header then select JSON since it is the first item in the list
    - produces={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}), // Output
    - consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE}) } // Input
  - if media type is not in list then sends an exception

Spring MVC – Rest Template

- Provides higher level methods that correspond to each of the main HTTP Methods
- Name Convention : Http Verb + what is being returned
  - getForEntity example
- Object passed and returned are convert to and from HTTP Message
- It will access a running
- If you are not interested in the entire ResponseEntity, including headers and status code, using getForObject will directly extract the payload out of the underlying ResponseEntity for you without you extracting it yourself via the method getBody() on the ReponseEntity
- Example
  - Public <T> getForObject(String url, Class<T>, responseType Object…, urlVariables) throws RestClientException
  - public <T> ResponseEntity<T> getForEntity(String url, Class<T>, responseType Object…, urlVariables) throws RestClientException)
  - Retrieve a representation by doing a GET, extract the payload from the ResponseEntity and returns the payload object through a converter
  - The parameter of for this method are:
    - URL
    - The Type of the return value are
    - Varargs of url variables
    - It will return the converted object

Spring MVC – Rest Post

- Post do not return data
- Calling Code
  - @GetMapping(path=”{id}”, produces={MediaType.APPLICATION_JSON_VALUE})
  - public ResponseEntity<Employee:> = get(@PathVariable(“id”) long id) {}

Spring MVC – Content Negotiation

- Dictate the format of the response
- Server Driven Negotiation – If the selection of the best representation for a response is made by an algorithm located at the server
  - Looks for an accept header first.
- agent drive content negotiation –
- By default a service returns json based of the Spring boot classpath, but using the produces attribute we can change that.
  - Request with a specific accept header can effectively ask for what they ant back
  - If no accept the default is Json
- Use JaXB requires a root element so we have to annotate the employee class
- if you send an accept header with media types that obtain no match with the service endpoint mediaTypes you get an 406 Response
  - If there is a match we use that header
  - order matters
- Problem : When you have a collection and return an Array List. A Java Collection classes are not annotated with @XmlRootElement
  - use Json since it wraps a collection
  - Create a class to wrap the XML

Spring MVC – Content Negotiation

- The client of a service can request what the textural response type it requires by sending an “Accept” header with the desired MediaType. If no Accept header is sent, the service will decide on the textural response format. In addition, posting data to a service, the service can identify endpoints that only consume a specific textual format by consuming a request with a specific content-type header

Spring MVC – Content Negotiation

Sending a content-type header to match to the consumes attribute

```
HttpHeaders headers = new HttpHeaders();
headers.add("accept", MediaType.APPLICATION_XML_VALUE);
headers.add("content-type", MediaType.APPLICATION_JSON_VALUE);
Employee empl = new Employee(99, "Fred", "Flintstone");
ResponseEntity<String> responseEntity = new
RestTemplate().postForEntity("http://localhost:8081/work/employee", new
HttpEntity<Employee>(empl, headers), String.class);
```

```
@PostMapping(consumes={MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<String> addStudent(@RequestBody Employee employee) {
    long id = dao.add(employee);
    if(employee.getId() > 0) {
        URI Student. Class = URI.create("/employee/staff?id=" + id);
        return ResponseEntity.accepted().location(uri).build();
    } else {
        return ResponseEntity.badRequest().build();
    }
}
```

Spring MVC – Content Negotiation

- So, return Json from such an endpoint or wrap the Collection in a class that you can annotate with @XMlRootElement

```
@XmlRootElement
public class EmployeeCollection {

private Collection<Employee> employees;
//getter and setters not shown
}
```

```
@GetMapping(produces=MediaType.APPLICATION_XML_VALUE)
public EmployeeCollection getAll() {
    EmployeeCollection emplCol = new EmployeeCollection();
    emplCol.setEmployees(dao.getAll());
    return emplCol;
}
```

```
<employeeCollection>
    <employees>
        <firstName>Eric</firstName>
        <id>1</id>
        <lastName>Colbert</lastName>
    </employees>
    <employees>
        <firstName>Mary</firstName>
        <id>2</id>
        <lastName>Contrary</lastName>
    </employees>
    <employees>
        <firstName>Jason</firstName>
        <id>3</id>
        <lastName>Ross</lastName>
    </employees>
    <employees>
        <firstName>Eric</firstName>
        <id>4</id>
        <lastName>Stewart</lastName>
    </employees>
</employeeCollection>
```

Spring MVC – Content Negotiation

Content Negotiation – Setting Headers for XML

```
HttpHeaders headers = new HttpHeaders();
headers.add("Accept", MediaType.APPLICATION_XML_VALUE);
ResponseEntity<String> responseEntity =
new RestTemplate().exchange("http://localhost:8081/work/employee/staff" + "?id={id}",
HttpMethod.GET, new HttpEntity(headers), String.class, 1);
System.out.println(responseEntity.getBody());
```

```
@GetMapping(path="/staff",
produces= {MediaType.APPLICATION_JSON_VALUE, MediaType.APPLICATION_XML_VALUE})
public ResponseEntity<Employee> getOne(@RequestParam("id") long id) {
    Employee empl = dao.getOne(id);
    if(empl == null) {
    return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok().body(dao.getOne(id));
}
```

```
<employee><firstName>Eric</firstName><id>1</id><lastName>Colbert</lastName></employee>
```

Can also set header for JSON

For multiple have acceptable formats have

heads.add(“Accept”, “application/json”, application/xml”);

Spring MVC – Rest Template

- Example
  - String url = “http://localhost:8091/work/employee/{id}”
  - String string = new RestTemplate().getForObject(url, String.class, 1)
  - System.out.println(string);

```
@GetMapping
public ResponseEntity<Employee> get(@PathVariable("id") long id) {
    Employee empl = dao.getOne(id);
    if(empl == null) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok().body(dao.getOne(id));
}
```

```
{"id":1, “firstName”:“Eric”,"lastName":“Colbert"}
```

- Get for Entity Example
  -

```
String url = "http://localhost:8081/work/employee/{id}";
ResponseEntity<String> entity = new RestTemplate().getForEntity(url, String.class, 1);
System.out.println(entity.getStatusCode());
System.out.println(entity.getBody());
```

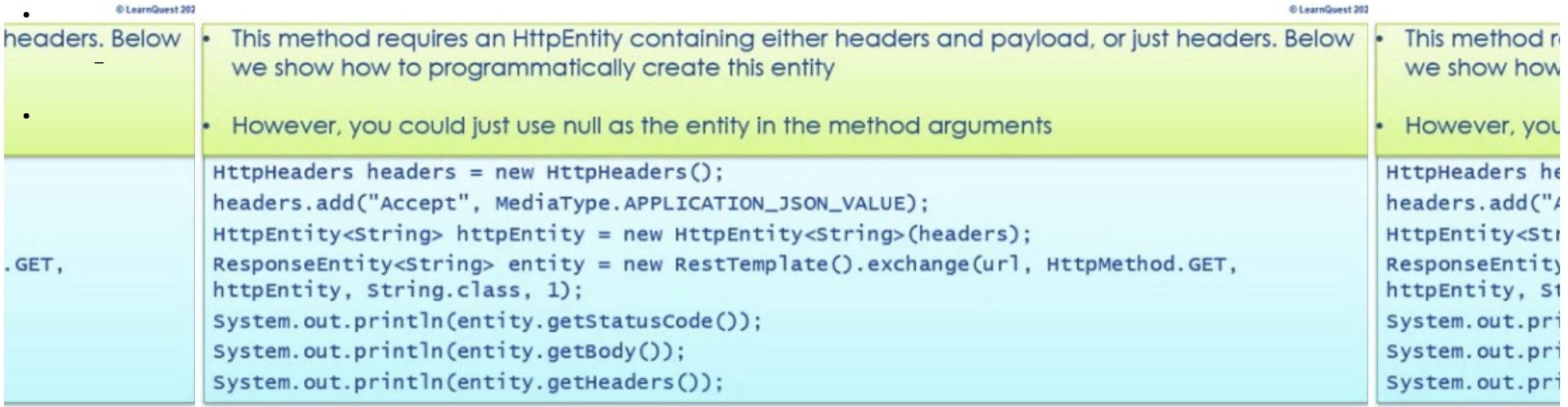
```
@GetMapping
public ResponseEntity<Employee> get(@PathVariable("id") long id) {
    Employee empl = dao.getOne(id);
    if(empl == null) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok().body(dao.getOne(id));
}
```

```
200 OK
{"id":1, “firstName”:“Eric”,"lastName":“Colbert"}
```

Spring MVC – Rest Template

Public <T> ResponseEntity<T> exchange(String url, HttpMethod method, HttpEntity<?> requestEntity, Class<T> responseType, Map<String, ?> urlVariables) throws RestClientException

Execute the HttpMethod to the give URI Template written the given request entity to the request and return the response as ResponseEntity
Example



Spring MVC – Rest Template

- Marshal directly into a java object other than a String
-

```
ResponseEntity<Employee> responseEntity =
new RestTemplate().exchange("http://localhost:8081/work/employee/staff" + "?id={id}",
HttpMethod.GET, null, Employee.class, 1);
assertThat(responseEntity.getStatusCode(), equalTo(HttpStatus.OK));
System.out.println(responseEntity.getBody().getFirstName());
```

```
@GetMapping("/staff")
public ResponseEntity<Employee> getOne(@RequestParam("id") long id) {
    Employee empl = dao.getOne(id);
    if(empl == null) {
        return ResponseEntity.badRequest().build();
    }
    return ResponseEntity.ok().body(dao.getOne(id));
}
```

Eric

Spring MVC – Rest Template

- Returns the collection of data
- Implementing a ParametizedTypeReference is an interface and is being used to give the routine the generic (type of what you want)
  - Allows the routine to build the collection

A ParameterizedReference can be used to capture a generic Type and retain it at runtime in order to obtain a Type instance

```
ResponseEntity<Collection<Employee>> responseEntity =
new RestTemplate().exchange("http://localhost:8081/work/employee", HttpMethod.GET, null,
new ParameterizedTypeReference<Collection<Employee>>() {
});
assertThat(responseEntity.getStatusCode(), equalTo(HttpStatus.OK));
responseEntity.getBody().forEach(p-> {
    System.out.println(p);
});
```

```
@GetMapping
public Collection<Employee> getAll() {
    return dao.getAll();
}
```

```
Employee [id=1, firstName=Eric, lastName=Colbert]
Employee [id=2, firstName=Mary, lastName=Contrary]
Employee [id=3, firstName=Jason, lastName=Ross]
Employee [id=4, firstName=Eric, lastName=Stewart]
```

Spring MVC – Rest Template

- Posting Data

```
Employee empl = new Employee(99, "Fred", "Flintstone");
ResponseEntity<String> responseEntity = new
RestTemplate().postForEntity("http://localhost:8081/work/employee", new
HttpEntity<Employee>(empl), String.class);
String locationUrl = responseEntity.getHeaders().get("location").get(0);
ResponseEntity<Employee> response = new
RestTemplate().getForEntity("http://localhost:8081/work/" + locationUrl,
Employee.class);
System.out.println(response.getBody());
```

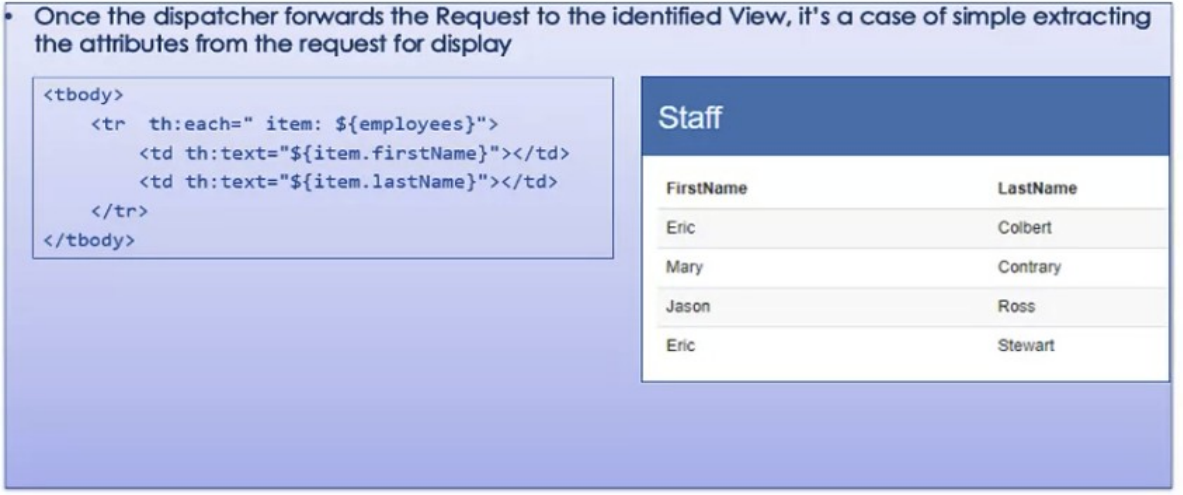
```
@PostMapping
public ResponseEntity<String> addStudent(@RequestBody Employee employee) {
    long id = dao.add(employee);
    if(employee.getId() > 0) {
    URI Dao. Add = URI.create("/employee/staff?id=" + id);
        return ResponseEntity.accepted().location(uri).build();
    } else {
        return ResponseEntity.badRequest().build();
    }
}
```

Controllers and View Resolvers

- @Controller classes have their methods mapped to URLs and Http methods in order for the Dispatcher servlet to delete to them
  - The return is interpreted as a destination (view) that is passed to a viewResolvers via Spring Dispatcher
- Although we can inject the HttpRequest into a contorller and set an attribute as below
  - Example
    - @GetMapping
    - public String getAll(HttpServletRequest request) { Couples the method to an HTTP Protocol instead Spring Provides a
      - request.setAttributes(“employees”, dao.getAll()); middleman calls the model – A glorified map that the DispatcherServlet
      - return “home” create before delegating a controller mathod
    -
    - If we put attributes in an injected model, the DispatcherServlet effectively takes these attributes and place them in the request object as Attributes. A request that is forward to a destination or view and is availabe to have attributes extracted from it
    -
    - @GetMapping
    - public String getAll(Model model) {
      - model.addAttribute(“employees”, dao.getAll());
      - return “home”;
      - }
    - Since we populated the request it does a server side forward to the object

Controllers and View Resolvers

- Thyme Leaf – Use the view
- Spring Boot
  - src/main/resources/templates
  - src/main/resources/static
  - src/main/resources/css
- Can have a ModelAndView
  - mv.addObject
  - mv.setViewName
- Can even patss the model in and the ModelAndView would know what to do.



Thymeleaf Templates

- A java template engine which is an alternative to JSP
- A natural template engine meaning you can view templates in your browser
- Example
  - <html lang=”en” xmlns:th=”http://www.thymeleaf.org”> // Specifying the Thymeleaf namespace
  - <body>
    - <table>
      - <tr>
        - <th>ID</th>
        - <th>Title</th>
        - <th>Author</th>
      - </tr>
      - <tr th:each=”book : ${books}”>
        - <td th:text=”${book.id}”>123</td> // Show when the template is the url in the browser
        - <td th:text=”${book.title}”>Spring in Action</td>
        - <td th:text=”${book.publisher.name}”>Wrox</td>
      - </tr>
    - </table>
  - </body>

Introduction to Pet Clinic
Running Spring Pet Clinic
Spring Pet Clinic

- A reference application to use the Spring Framework
- Allow Owners / Veterinarians to schedule appointments
- Github spring-projects, spring-petclinic-community
- Running Spring Pet Clinic
  - Spring-projects / spring-petclinic
  - .mvnw spring-boot:run
- Spring Pet Clinic
  - Spring Boot – Brings the number of Opinionated default configurations
    - Save you time since you do not have to research the best way to do it
  - SFG (Spring Framework GURU) Pet Clinic will also be opinionated software development
    - Best Practices in Software Design : OOP/SOLID, TDD, Naming Conventions, Software Development in Life Cycle
    - Mainline dev model, use branching moderately
    - Use Github Issues to plan work and tasks

Spring Pet Clinic – Initializing Spring PetClinic Application

- Create New Repository
  - Use Git Hub and select Create New Repository
  - Provide Name
    - Make Repository private until you are ready to release it.
    - Can set it up to initialize with readme
    - Add .gitignore
    - Add a license
  - Click on Creating Repository
- Creating a project in Intellij
  - Select New Project (Spring Initializer)
    - Can use the web version which will download a zip file
    - Add the project metadata data such as coordinates, Artifact, Type, Language, Packing, Java Version, Version, Name, Description, Package
    - Select Dependencies such as SQL, NOSQL, JTA, Cloud Core etc…
    - Select Project Name, Project Location, module name, content root, module file locations, project format
- Intellij – enable auto changes could be a problem for large project
- Borrow his .gitignore files

Spring Pet Clinic Task Planning

- GitHub has an issue tracker
- Business partners create an issue.
  - Example Credit card companies do not change the code unless you have an issue which is useful for traceability
- From GithHub Click Issue
  - Can create a new issue by clicking the new issue button

Solid Principles of OOP

- Why use the Solid Principles of OOP
  - OOP is a powerful concept, but does not always lead to quality software
  - 5 principle focus on dependency management
    - Poor dependency management leads to code that is brittle, fragile and hard to change
    - Good dependency management lead to quality code that is easy to maintain and easier to test
- Principles
  - Single Responsibility Principle Just because you can does not mean you should
    - Every class should have a single responsibility
    - There should never be more than one reason for a class to change
    - Your classes should be small. Nor more than a screen full of code
    - Avoid “God Classes”
      - Classes with 2000 lijnes functions are very hard to test.
    - Split big classes into smaller classes
  - Open Closed Principal
    - Your classes should be open for extension, but closed for modification
    - Should be able to extend class behavior without modifying it
    - Use private variables with getters and setters only when you need them
      - Don’t expose every property only the ones that you need
    - Use abstract base classes

Solid Principles of OOP

- Liskov substitution principal
  - Objects would be replaceable with instances of their subtypes without altering the correctness of the program
  - Violations will often fail the ISA Test
    - Example: A Square iSA Rectangle True
    - Example: A Rectangle ISA a Square False
- Interface Segregation Principle
  - Make fine grained interfaces that are client specific
    - Many client specific interfaces are better than one general purpose interface
  - Keep your components focused and minimize dependencies between them
  - Notice the relationship to the Single Responsibility Principle
    - i.e. avoid ‘god interfaces’
    - Avoid hug general purpose one
- Dependency Inversion Principle
  - Abstractions should not depend upon details and details should not depend on abstractions
  - Important that higher level and lower level object depend on the same abstract interaction
    - Should not share a dependency , but have a clearly defined API
  - Not the same as Dependency Injection (which is how objects obtain dependent objects)

Create an Spring Framework DI Example Project

- Create a Spring Project using Intellij
  - New → Project → Spring Initializer
    - Provide Java SDK
    - Choose initalizr service (URL)
    - Maven coordinates
    - Type
    - Java Version
    - Language
    - Name
    - Description
    - Package
  - Select all the dependencies
- Intellij indexes the project
- If you see anything weird then select Files → Invalidate Caches
- Best Practice : For logging use SLFJ and not a hard logging

Inversion of Control

- Inversion of Control
  - Programmers don’t create their objects, they need to be describe how they should be created
  - application code does not directly connect your implementation components together
- Loose Coupling through interfaces
  - Using Brittle Instances with tight coupling
- Inversion of Control (IoC)
  - A technique to allow dependencies to be injected at runtime
  - Dependencies are not predetermined
    - Some type of framework to determine what dependencies to inject

Spring Component Scan

- Purpose : Using Component Scan to look at the package structure and find Spring Bean. Looking for @Configuration or stereotypes.
- @SpringBootApplication – Do a component scan from the current package to the rest of the children packages.
- Example
  - Package Path guru which has two packages services and springframework and springframework has controller
  - package guru.springframework
  - @SpringBootApplication
  - @ComponentScan(basePackages = {“guru.services”, guru.springframework }) // Add guru.services to the component scan
    - When using basePackage overwrote the default behavior of starting at guru.springframework
    - It will only look in the packages that you specify if basePackages is being used.

The Spring Framework Context

- Stores the Beans that are created
- Keep track of the dependencies
- SpringApplication.run returns an ApplicationContext
  - Method getBean(“myController”); // Creates the bean with the name myController
- org.springframework.beans.factory.BeanFactory interface to create and manage beans
  - The BeanFactory implementation is the actual container which instantiates/configures and manages a number of beans through it API
  - Functions getBean, containsBeans, getAliases
- ApplicationContext – build on top of bean factory and add Functionality
    - Easier integration with AOP
    - Message resource handling (ex. internationalization)
    - Event propagation
    - Declarative Mechanisms to create ApplicationContext
    - Parent Context
    - Application Layer specific context such as WebApplicationcontext

How To Request a Java Object

- Configuration Read
- Bean Definitions Created
- Java Object Created
- Application Context Created
- Request made using Bean Definition Id
- Code to the interface
  - instantiates an ApplicationContext interface to represent the IoC Container – need an XML File
  - Bean definitions are create by the configuration then Spring Managed Bean instances are created eagerly from the bean definitions
    - cached in the Ioc Container
  - Ex.
    - EmployeeServer employeeService = (new ClassPathXmlApplicationContext(“beans.xml”)).getBean(“emplService”, EmployeeService.class);
- Using Value
  - <bean id=”emplService” name=”employeeServ” class=”com.employeeRus.servic.EmployeeServiceImp”>
    - <property name=”dao” ref=”emplDao” />
    - <property name=”active”, value=”true” /> // Can be a String or Boolean
    - <property name=”mode”, value=”true” />
  - </bean>

The Spring Framework Context

- How do it create one
  - Read in the meta to create the bean definition
    - looks in the ClassPath for the configuration file – ClassPathXmlAppicationContext
    - looks in the File System – FileSystemXmlApplicationContext
    - looks in the Web – WebXmlApplicationContext
- Code
  - ApplicationContext appContext = new ClassPathXmlApplicationcontext(“beans.xml”);
  - EmployeeService employeeService = context.getBean(“emplServiceImpl”, EmployeeService.class) // EmployeService is an interface
    - Change the class then you do not need to change the code
- ClassPathXmlApplicationContext
  - Uses Spring Schemas
  - Root Element is tag beans and bean defined individual bean
  - schema-instance namespace defines the tags used within the schema
  - <bean id=”emplService” name=”employeeServ serv,/service” class=”com.emplyeeRus.service.EmployeeServiceImpl” />
    - For the name we have two aliases : employeeServ serv, /service
- Dependency Injection for XML Configurations : Field
- Spring Managed Bean are eagerly created

Basic of DI

- A needed dependency is injected by another project
  - The class being injected has no responsibility in instantiating the object being injected.
    - The class is only stating that it needs the object. It is the responsibility of the Spring Framework to instantiate the object and then inject it.
- Some say you avoid declaring objects using ‘new’
  - Example a database connection (hardwiring it means you need to close connection, get configuration)
    - What if you want to move to a new database server
  - Example – Put a message on a queue
- Types of Dependency Injection
  - By class properties
    - Least preferred
    - Using private properties is evil – Cannot unit test that class
    - If Fails your DI will have a corrupt object create by Spring, but with invalid property. Being private your code cannot even bring the object to a proper state by manually assigning the required dependency.
- In the Configuration everything is a string and PropertyEditor converts to the correct value

Basic of DI

- By Setters
  - Have a chance of not setting the dependency
  - <bean id=”emplService” name=”employeeServ” class=”com.employeeRus.service.EmployeeServiceImpl>
    - <property name=”dao” ref=”emplDao” /></bean>
  - <bean id=”emplDao” class=”com.employeeRus.dao.EmployeeDaoImpl” />
  - Spring Style
    - Our bean tag has defined a BeanDefinition which upon loading the ApplicationContext will be parsed into a BeanDefinition Object
    - These definitions are used by the IOC Container to crate managed bean instances
    - Now use Setter Inject
      - The property dao looks for a setter called setDao and tries to inject in the bean whose id is emplDao
      - If not found you get an UnsatisitededDependencyException
    - code
      - public class EmployeeServiceImpl implements EmployeeService {
        - private Employee dao; public void setDao(EmployeeDao dao) { this.dao = dao; } }
        - setDao injects the dao
- By Constructor
  - Pure Object Orientated prefer by Constructor – Can’t instantiate without the dependencies
  - <bean id=”emplService” name=”employeeServ” class=”com.employeeRus.service.EmployeeServiceImp”>
    - <constructor arg=” ref=”employeeDao” /><constructor-arg value=”true”><constructor-arg value=”3”> // ref – refers to a bean definition
  - can use name attribute to resolve it to the parameter value and type or can use index attribute which is 0

- A pattern used to create instance of objects that have delegation patterns to other objects at compile time
  - The calling class has No knowledge of the dependency class implementation it will delegate at Runtime
  - Interfaces

Basics of DI

- Concrete classes vs interfaces
  - DI can be done with Concrete Classes or with interfaces
  - Generally DI with Concrete classes should be avoided
    - Stating that you must have this specific class and Dependency Injection is support to put in the correct class (ex. Database class for tier)
  - DI via interfaces is highly preferred
    - Allow runtime to decide implementation to inject
    - Follows Interface Segregation Principle of Solid
    - Makes your code more testable
      - Can write your own mocks, can use mockito and do true unit test
      - With Concrete you will have to inject the exact object
        - Example Inject a MySqlDatabaseConcreteCllass → Inject mysql datatype
-
  -

Basics of DI

- The methods defined by the user to tailor the framework will often be called from within the framework itself, rather than from the user's application code.
- The framework plays the role of the main program in coordinating and sequence application activity.
- This inversion of control gives frameworks the power to serve as extensible skeletons. The methods supplied by the the user into the generic algorithms define in the framework for a particular application

- Ioc vs Dependency Injection
  - DI refers much to the composition of your class
    - You compose your class with DI in mind
  - IoC is the runtime environment of your code
    - Spring Framework's IoC Container
- If there is no id attribute it will use a generated name
  - full qualified class name with # and occurrence instance
  - If a name attribute and no id spring will interpret the name as the bean'sid and not generate one
-

Dependency Injection with the Spring Framework

- Example
  - Application ctx = SpringApplication.run(DiDemoApplication.class, args);
  - MyController controller = (MyController)ctx.getBean(“myController”);
  - controller.hello();
  - System.out.println(ctx.getBean(PropertyInjectionController.class).sayHello());
  - System.out.println(ctx.getBean(GetterInjectionController.class).sayHello());
- Error : No qualifying bean of type guru.springframework.controllers.PropertyInjectorController available
  - The class did not have the annotation to create the bean. Did not have @Controller, @Configuration, @Service
- Error : Null Pointer Exception at Property Injected Controller
  - Did not tell Spring to inject the class ex. @autowired
  - This error means unsatisfied dependency
- Error – Parameter <n> of method <fully qualified class name> required a bean of type <fully qualified clas name> that could not be found
  - Make sure the class has an annotation allows for the class to be created as a Spring Bean
- Constructor Injection after 4.2 enable automatic (@autowired) – enable automatic wiring of constructor components
  - However it is ok to add it

Dependency Injections – Annotations

- annotations config
  - <context:annotation-config />
- @autowired – Constructors, fields and setter methods
  - A method with an arbitrary name and arguments which all arguments with a matching bean
  - Collections or an array where it injects all bean of a type
  - Example
    - @Component
    - public class EmployeeServiceImpl implements EmployeeService
    - {
      - @Autowired
      - public void setDao(EmployeeDao dao) { this.dao = dao; }
      - @Value(“true”)
      - private String active
    - }
    - @inject is similar to autowired and @name is similar to Qualifier. Important since the default scope is prototype – It is javax
      - Can use PostConstruct to initialize simple values – Java Annotation – Week 2 Spring Course Ra

Using Spring Qualifier Annotations

- What if you have more than one implementation of an interface.
  - One of the tools is qualifier annotation – tell spring which one we want.
- Problem : You have interface : GreetingService which has two implementations : ConstructorGreetingService and PropertyGreetingService
  - When running without the Qualifier get an error
- Error : Parameter <n> of constructor <fully qualified class name> required a single bean, but 3 were found
  - List all beans the 3 implementations of the GreetingService Interface
  - Either give the name of the class or the name set with @Qualifier when creating the bean.
  - Fixed for the Constructor Injection use – Only one specific bean is selected) can be the name of the @Component class or give the @Qualifier(<name>")
    - Public ConstructorInjectedController(@Qualifier("constructorGreetingService") GreetingService);
    - @Autowired @Qualifier("getterGreetingService") Public void setGreetingService(GreetingService greetingService) at the method level or
    - @Autowired Pubic void setGreetingService(@Qualifier("getterGreetingService") GreetingService greetingService)
- The Qualifier allows use to specify the class name
- Error : Parameter <n> of constructor <fully qualified class name> required a single bean, but 3 were found
  - Could happen if the @Qualifier is not present or the property being injected is changed from a function to an interface
- Change Property Name to match the bean name (camelCase) – Spring will do reflection and see the bean name matches the property name
  - Be careful could cause unintended side effects

Primary Annotation for Spring Beans

- @Primary annotation – Have multiple beans of the same type, but one of them to go in by default
- Example
  - @Service // Brings it in as a Spring Bean
  - Public class PirmraryGreetingService implements GreetingService {return “primary service”}
  - @Controller
  - Public class MyController {
    - Private MyController(GreetingService greetingService) {this.greetingService = greetingService;)
    - Public String hello() { System.out.println(“Hello”); }
  - }
- When you run this without the @primary the following error is returned : Error : Parameter <n> of constructor <fully qualified class name> required a single bean, but 3 were found
  - List all beans the 3 implementations of the GreetingService Interface
- When you add the @Primary to the PrimaryGreetingService class and the class is the bean name then specifying the bean name is the best practice.
-

Spring Profiles

- Example
  - @Service
  - @profile(“es”)
  - Public static class PrimarySpanishSpeakingService implements GreetingService {
    - @Override
    - Public String sayGreeting() { System.out.println(“Hello in Spanish”);
  - }
  - @Service
  - @profile(“en”)
  - Public static class PrimaryEnglishSpeakingService implements GreetingService {
  - @Override
  - Public String sayGreeting() { System.out.println(“Hello in English”);
  - }
  - In the properties file add spring.profile.active = es // Makes the profile for Spanish active and Spring will use it
  - Control Feature Sets example activeMQ for development and rabbitMQ on production

Default Profile Behavior for Spring Context

- Special Profile call for default – only active when other profiles are not
- Have the English Greeting come out when there is not active profile that will meet the criteria.
- When all the beans have an active profile, there could be a case where there is no active profile
- Example
  - @Servrice
  - @profile(“de”)
  - Public static class PrimarySpanishSpeakingService implements GreetingService {
    - @Override
    - Public String sayGreeting() { System.out.println(“Hello in spanish”);
  - }
  - @Service
  - @profile(“en”, “default”) // Active if no profile is selected or en profile is selected
  - Public static class PrimaryEnglishSpeakingService implements GreetingService {
    - @Override
    - Public String sayGreeting() { System.out.println(“Hello in Spanish”);
  - }
- If you do not have an active profile then having more than 1 implementation which will cause an error where a single bean was required, but three were found.

Spring Bean Life Cycle

- Spring Bean Life Cycle – Getting Bean ready for use.
  - Instantiate — Constructor of injected class is called
  - Populate Properties — set values from interface
  - Call setBeanName of BeanNameAware — Next three lines are the one that get called
  - Call setBeanFactory of BeanFactoryAware
  - Call setApplicationContext of ApplicationContextAware
  - Preinitialization (Bean PostProcessors)
  - AfterPropertiesSet of Initializing Beans
  - Custom Init Method
  - Post Initialization (Bean Post Processors)
- Spring Bean Life Cycle : Container Shutdown
  - Get a container shutdown signal
  - Disposable Bean’s Destroy
  - Call Custom Destroy Method — Clean up resource such as a socket

Spring Bean Life Cycle

- Callback interfaces
  - Spring has two interfaces you can implement for call back events
    - InitializingBean.afterPropertiesSet()
      - Called after properties are set
    - DisposableBean.destroy()
      - Called during bean destruction and shutdown
- Life Cycle Annotations
  - Spring has two annotations you can use to hook into the bean life cycle
    - @PostContruct → Called after bean has been constructed, but before it returned to the requesting object
    - @PreDestroy →called just before the bean is destroyed by the container becoming non spring managed
- Bean Post Processors – The Teacher never found a use for these
  - Give you a means to tap into the Spring context life cycle and interact with beans as they are processed
  - Implement interface BeanPostProcessor
    - PostProcessBeforeInitialization – Called before bean initialization method
    - PostProcessAfterInitialization – Called after bean initialization
  -

Spring Bean Life Cycle

- Aware Interfaces
  - Spring has over 14 Aware Interfaces
  - These are used to access the Spring Framework infrastructure
  - These are largely used in the framework
  - Rarely used by Spring Developers
  - Todo : Find a list of them on the net
  - Application Context Aware
    - ApplicationContextAware — Gets the Application context handle
    - ApplicationEventPublisherAware — Setup Event Listener
    - BeanFactory Aware — Get a handle on the Bean Factory
      -

Spring LifeCycle

- Spring Beans have an extensive lifecycle.
- Constructor Injection happens before setter injection
- If the Spring Managed Bean implements the interface InitializingBean, we have an initializer method that will trigger after Setter Injection.
- BeanPostPorcessor interface – postProcessAfterInitialization. postProcesssBeforeInitialization

Spring Bean Lifecycle Demo

- Has a class that outputs messages for each output (should be executed on your own machine)
- Example → Called for each bean in the context and does something to it
  - Public class CustomPostProcess implements BeanPostProcessor {
    - @Override
    - Public Object postProcssingBeforeInitialization(Object bean, String beanName) {
      - If (bean instanceof LifeCycleDemoBean) {
        - ((lifeCycleDemoBEan)bean).beofreInit();
      - }
      - Return bean; }
    - @Override
    - Public Object postProcessAfterInitialization(Object bean, String beanName) throws Bean Exception {
      - If (bean instanceof LifeCycleDemoBean) {
        - ((LifeCycleDemoBean)bean).afterInit());
      - }
      - return bean;
    - }

Single Responsibility Principle

- Cohesion
  - How much the code segments within one module (method of a class, classes inside a package) belong together.
  - The higher the better
    - Easier maintenance and debugging
    - Greater code functionality and reusability
  - Loose Coupling is related to high cohesion
- Robustness
  - The ability to handle mistakes or malfunctions elegantly
  - Different ways to implement
    - Test the code with different inputs
- In order to achieve robustness and high cohesion
- Single Responsibility Principal
  - A single code module should only have responsibility over one part of the functionality provided by the software
  - Should have only one reason to change
  - A division of concerns is preformed in the program and the methods for for every concert should be encapsulated by a single class
    - Cohesion → methods related to the same concert will be members of the same class

Flash Cards

- Dependency Inversion addresses abstractions while dependency injection refers to the injection of Dependencies into a class
- Good idea to use concrete class : No you should use interface which will allow the runtime environment to determine the implementation to inject.
- IoC → The Runtime environment which injects dependencies
- Two callback interfaces you can implement to tap into the bean lifecycle : InitializingBean , DisposableBean
-

Single Responsibility Principle

- Founded on the idea of OOP
  - Divide and Conquer – solve the problem by solving its subproblems
  - Prevents the creation of “God Objects” and object that “Know Too much or do too much”
- Bad Example
  - A class with getters and setters and two member functions : findSubTextAndDelete(s), printText(), allLettersToUpperCase()
  - Multiple reason to change : two methods which change the text itself and one that print the text for the user
  - Mixes logic with the presentation
- Better Example
  - One way to fix this is create a class that is concerned with printing text
  - Have two class concerns : printText, modifyText
    - A class with getters and setters and two member functions : findSubTextAndDelete(s), allLettersToUpperCase()
    - A class that prints the text
- Make it easier to see the big picture
- Spring
  - As you become more comfortable with IoC and Dependency Injection your classes will naturally adhere to the the single responsibility principal.
  - Example JDBC in controllers → The controllers methods should be very simple and light. Database calls and other business logic belong in the service layer

Open Closed Principle

- Should extend the functionality by adding new code to meet the new requirements
- Open by Extension
  - Does not necessarily mean inheritance
  - Module should provide extension points to alter its behavior
    - Example polymorphism
- Closed for modification – The source code for such a module remains unchanged.
- Create abstractions that are fixed and represent an unbound group of possible behaviors through concrete sublcasses
- Bad Example
  - Problem : Insurance Systems that validates health insurance claims before approving one.
  - Follow Single Responsibility Principal :
    - HealthInsuranceSurveyor isValidClass() validate claims and ClaimApprovalManager to approve claims
  - Next we worry about VehichleInsuranceSurveyor, but we need to modify claminApprovalManager by adding processVehicleClaim()
  - We violated the code in two when we add processVehicleClaim to the ClaimApprovalManager
  - However we created the ClaimsApprovalManager class that needs to keep pace with fast changing business demand
    - For each change your to build/test/deploy the appliation all over again.

Open Closed Principle

- Good Design
  - Create an abstract class: InstanceSurveyor
    - IsValidClaim()
    - HealthInsuranceSurveyor which extends InstanceSurveyor
    - VehicleInsuranceSurveyor extends InstanceSurveyor
    - ClaimApprovalManager
      - Should contain a function processClaim(InsuranceSurveyor surveyor)
-

Listov Substitution Principle

- If S is subtype of T then object of type t may be replaced with object of type s without alternating any of the desirable properties
- There are three types
  - Inheritance
  - Polymorphism
    - Context dependent behavior – can behave one way in a certain situation and another in way in some other situation
      - The difference is not strict.
    - Example : A mother at school function or out with friends
  - Subtyping (subtyping polymorphism or inclusion polymorphism)
    - A form of type polymorphism in which a subtype is a datatype that is related to another datatype by some notion of sustainability mean the program elements typically subroutines or functions written to operate on elements of the supertype can also operate on elements of the subtype
    - If S is a subtype of T
      - The Subtype relation is written S<:T
        - Any term of type S can be safely used in a context where a term of type T is expected
- Problems
  - Violation :example A bicycle is a transportation devicem but it does have an engine. The method startEngine cannot be implemented.
    - Solution : Correct Inheritance Hierarchy have a TransportationDevice with and without an engine.
- If you write object which extend classes, but fails the IS-A Test you are likely violating the principle

Interface Segregation Principle

- Clients should not be forced to depend on methods that they do not use.
- Solution : Segregate a fat interface into smaller and highly cohesive interfaces know as roles interfaces
- Violation of the interface segregation principle also leads to violation of the complementary “Open Closed Principle”
  - The fat interface will never be closed You will always be adding a routine to it.
- The classes with cohesive interfaces will be easier to read and less cluttered.
- Have the same goals as the Single Responsibility, but is meant for interface

Dependency Inversion Principle

- When creating an object with new your are creating a class tightly coupled to another class
- When on class knows the internals of another class then the risk of break the class rises making the application fragile
  - Good Code is writing loosely coupled code you can turn to the Dependency inversion principle
- High level modules should not depend on low-level modules Both should depend on abstractions
- Conventional Architecture follows a top-down design aporach were a high-level and as a result high level modules get written directly depends on the smaller (low-level) modules
- Dependency inversion principle say instead of a high-level module depending on a low-level module both should depended on abstractions
- Abstractions should not depend on details. Details should depend on abstractions
- Example of Dependency Inversion Principle
  - Object A in Package A refers to Object B in Package B
  - Interface A is introduced as an abstraction in Package A
    - Object A now refers to Interface A
    - Object B inherits from Interface A
  - What this has done is
    - Both Object A and Object B now depends on Interface A the abstraction
    - It inverted the dependency that existed from Object A to Object B into B being dependent on the Abstraction (interface A)

Dependency Inversion Principle

- Example a program to turn a light bulb on or off
  - Class LightBulb with methods turnOn and turnOff
  - Electric Power Switch has methods isOn and press() and has a lightBulb
  - High level class ElectricPowerSwitch is dependent on the LightBulb
  - A switch should not be tied to a build it should be able to turn on/off other appliances
  - Solution : We will need an abstraction that both the ElectricPowerSwitch and LightBulb classes will depend on
    - Switch which is an interface has isOn() and press()
    - Switchable which is an interface has turnOn(), turnOff()
      - Each class that implements Switch can turn the device on or off the way it wants to
  - ElectricPowerSwitch will implement Switch and contains a client that implements the Switchable phase
  - LightBulb Implement the Switchable
- Summary of the Dependency Inversion Principle
  - First class combination of the Open Closed Principle and the Liskov Substitution Principle
  - Dependency Inversion is more focused on the structure of your code keeping the code loosly coupled
  - Dependency Injection is how the code functionally works
    - Spring uses Dependency Injection to build your application.

Interface Naming Conventions

- Interface Naming Conventions
  - Interface should be a good object name
    - Examle Java List Interface
      - Example Implementations : ArrayList, LinkedList, CheckedList, SignletonList
    - When you provide the interface name your not worried about the nitty gritty of the classes. Name your interfaces at a high level
  - Don’t Start with I ex. No Ilist
- Implementation
  - When just one implementation – generally accepted to use <interface name> + Impl
    - Don’t do this be cause one implementation becomes many in time.
  - When more than one name should indicate difference of implementation
-

Tips and Tricks – Custom Banner

- URL : patorjk.com and select “Text to ASCII Art Generator”
- Steps
  - Copy to clip board
  - In Resources create banner.txt and past the code into the file

Spring Pet Clinic – POJO Data Model
Spring Pet Clinic Mutli-Module Maven Builds

- Split a project into modules
  - Example Split out Data Class into its own Module
- Add a new Artifact ID → pet-clinic-id, Module name pet-clinic-data, Content root: sfg-pet-clinic/pet-clinic-data, sfg-pet-clinic/pet-clinic-data
- The Top Level will be sfg-pet-clinic and pet-clinic-data inside
  - Have the normal maven structure for each module
  - Added the files / dependencies into the correct maven module
- Intellij after a refactoring may require a refresh
- Intellij maven project need to be imported
- IN the sfg-pet-clinic has a tag <modules><module>pet-clinic-data</module></modules>
- Can make each module a jar
- By default Spring Boot will want to do a fat executable jar to include all the dependencies. In the POM File
  - In the pom file for <build><execution> use a goal <repackage> with a <configuration> skip = true
  - As of Spring 2.1 we have a propety for maven spring.boot.repackage.skip = true
- In a maven file when you have the pom file and no source files best to remove the source directory
- In Github if you have an issue and in the commit you state closes <number> will close out the github issue

Spring Pet Clinic – Maven Release Plugin

- Maven Release Plugin
  - Generate Releases of artifacts
    - mvn release:prepare – maven.apache.org/maven-release/maven-release-plugin/examples/prepare-release
      - Ask for a release version
    - mvn releaseperform – maven.apache.org/maven-relase/maven-release-plugin/examples/perform-release
  - scm tag → Describe the repository where the code is located
  - AutoVersionSubmodules → For the Two submodules it provides the same version for the submodules

Spring Pet Clinic – Create Interfaces for Services
Spring Pet Clinic – Implement Base Entity
Spring Pet Clinic – Using an image for Custom Banner

- Implement a Service Layer
  - Have a service layer interact with the repository
    - Provide different implementations of the services.
    - We can have a controller and inject a different object into the class
      - Lets say we have the owner interface contains CRUD for an owner
        - We could inject an Owner where the data is stored in a Map
        - We could inject an Owner where the data is stored in a relational database
  - Package → guru.springframework.sfgpetclinic.services
  - OwnerService
    - Contains findById(long id)
    - save(Owner owner)
    - findByLastName(Stirng lastName)l;
    - Set<Owner> findall();
- Implement Base Entity
  - Hibernate recommends box type (ex. Long) be used instead of primitive since they can use null
- Using an image Banner
  - Put the image in the resources
  - In Application Properties use spring.banner.image.loocation = <name of image>
    - The spring logo gets replaced with the picture

Spring Pet Clinic – Refactor Service to Common Interface
Spring Pet Clinic – Implement Map Based Services
Spring Pet Clinic – Using and Image for Custom Banner

- Create a CRUD Service that will take a type and an ID.
- Example
  - Public interface CrudService<T, ID> {
    - Set<T> findAll();
    - T findById(ID id);
    - T save(T object);
    - Void delete(T object);
  - }
- Implement Map Based Services – Produce a concrete Service Layer using Maps
  - Implement an AbstractMapService<T, ID> with findall(), findBy(ID), T save(ID, T), deletebyId(ID), delete(T Object)
    - Implement for Owner,
  - Example Code : map.entrySet().removeIF(entry → entry.getValue().equals(object)
- Using and Image for Custom Banner
  - See Spring Pet Clinic – Using an image for Custom Banner

Create Vet Index Page and Controller
Create Owner Page and Controller

- For the Root Page you call it index.html or a more specific name such as vets.html
- More Thyme Leaf
  - <h1 th:text="'List of Vets...'">
- In the Controller from listVets()
  - return a string “vets/index” which means look for the vets/index.html file
  - @RequestingMapping “/vets”, “/vets/index”, “/vets/index.html”
- Create Owner Page and Controller
  - For a class with @Controller
  - The @RequestMapping(“/owners”) at the class level forces every request mapping to become prefix with “/owners”
  - Also need an empty string the method level request mapping so we can have /owners

Spring Configuration Options

- Types
  - Xml Based Configuration
    - Introduced in Spring Framework 2.0
    - Common in legacy Spring Applications
    - Still supported in Spring Framework 5.x
    - Xml Schema very important since IDE(s) are very good at reading the schema
  - Annotation Based Configuration
    - Introduced in Spring Framework 3
    - Picked up via ‘Component Scans’
    - Refers to Class Level Annotations – @Controller, @Service, @Component, @Repository
    - Spring boot will find the @SpringBoot Annotation and then search the current package and child packages.
      - If the class is move out of the package/child package then a class not found error will happen.
  - Java Based Configuration
    - Introduced in Spring Framework 3.0
    - Use Java Class to defined Spring Beans
    - Configuration are with @Configuration and beans are declared with @Bean annotation

Spring Configuration Options

- Groovy Bean Definition DSL Configuration
  - Introduced in Spring Framework 4.0
  - Allow you to declare beans in Groovy
  - Borrowed from Grails (Groovy wrapper around Spring Framework) and usually used there

- Which to use – They will work seamlessly together to define beans in the Spring Context, the industry favors Java based configuration

Spring Framework Stereotypes

- Stereotype – A fixed general image or set of characteristics which represent a particular type of person or thing
- Spring Stereotypes are used to define Spring Beans in the Spring Context
  - When it scan a package it is looking for particular annotations (sterotype annotations)
    - Available Stereotypes are @Component, @Controller, @RestController, @Repository, @Service
      - Hierarchy
        - @Component is the Base class
        - @Controller, @Repository, @Service inherit from @Compoent
          - @controller maps a particular method to an URL
          - @Repsotiory – accessing the data layer
        - @RestController (Convenience annotation representing @Controller and @ResponseBody) inherits from @Controller
- Description of Stereotypes
  - @Component Indicates the annotated class will be created as a bean and @Component – indicates that a class fulfills the role of a stereotype
  - @Controller The annotated class has the role of a Spring MVC Controller
  - @RestController extends @Controller to add @ResponseBody
  - @Repository Encapsulating storage, retrieval and search behavior which emulates a collection of objects
  - @Service An operation offered as an interface that stands alone in the model with no encapsulated state
- Common Pattern – The Controller will have a service injected into the controller and the service will have a repository injected into the service

Example

- Example
  - @Component
  - public class AppConfig {
    - @Inject private EmployeeDao dao; // Comes from another configuration
    - @Bean(name=”service”);
    - EmployeeServiceImpl = getEmplService(@Value(“${min})“ int min) {
      - EmployeeService service = new EmployeeServiceImpl();
      - service.setDao(dao);
      - return service;
    - } }
  - Advantageous
    - Type Safe since it written in Java
    - Compile time checking

Spring Framework Stereotypes

- @Repository Error Handling
  - If you are using Hibernate/JDBC and different DB the exceptions from that layer and convert them into a Spring Exception so the error handling for the Spring Framework can stay generic
    - Instead of handling MySQL or Oracle Error you are handing a DataAccessException
- With stereotype usage you are trying to express intent

Java Configuration Example

- Example – Create a folder called config and include
  - ChuckConfiguration.java – Replace the constructor. The class is coming from an external jar
    - @Configuration
    - Public class ChuckConfiguration {
      - @Bean
      - Public ChuckNorrisQuotes chucksNorrisQuotes() {
        - ChuckNorrisQuotes test = new chuckNorrisQuotes();
        - test.setNumber(50);
        - Return chuckNorrisQuotes;
        - }
- We are hard wiring the class, but for the different environments we can include a different configuration directory
- @bean can have name and init-method

Spring XML Configuration

- Put the XML Configurations in the resources directory.
- For the beans we the following
  - http://www.springframework.org/schema/beans
  - http://www.springframework.org/2001/XMLSchema-instance
  - Schema Location : http://www.springframework.org/schema/beans http://www.springframework.org/schema/beans/spring-benas.xsd
- In the resource file for the file chuck-config.xmsl add the following
  - <beans attributes are shown above… >
    - <bean name=”chuckNorrisQuotes” class=”guru.springframework.norris.chuck.ChucksNorrisQuotes”></bean>
    - </beans>
- In the guru.springframework.joke package for JokeappApplication.java
  - @SpringBootApplication
  - @importResource(“classpath:chuck-config.xml”) // Search the class path for chuck-config.xml
  - Public class JobkesappApplication { Public static void main(String args[]) { SpringApplication.run(JokesApplication.class,args); } }
- ApplicationContext has overloaded constructors that take advantage of varargs of xml files

Using Spring Factory Beans

- Example
  - createGreetingLineService.java A factory class
    - @Component
    - public class GreetingServiceFactory {
      - public GreetingRepository greetingRepository;
      - public GreetingServiceFactory(GreetingRepository greetingRepository) { this.greetingRepository = greetingRepository; }
      - Public GreetingService createGreetingService(String lang) {
        - switch(lang) {
          - case “en”:
            - return new PrimaryEnglishGreetingService(greetingRepository);
          - case “fr”:
            - return new PrimaryFrenchGreetingService(greetingRepository);
          - default:
            - return new PrimaryEnglishGreetingService(greetingRepository);
        - }
      - }
    - }

Using Spring Factory Beans

- @Configuration
- Public class GreetingServiceConfig {
  - @bean
  - GreetingServiceFactory greetingServiceFactory(GreetingRepository repository) { return new GreetingServiceFactory(repository); }
  - }
  - @bean
  - @Primary
  - @Profile({“default”, “en”})
  - GreetingService primaryGreetingService(GreetingServiceFactory greetingServiceFactory) {
    - return greetingServiceFactory.createGreetingService(lang, “en”); }
  - @bean
  - @{Profile{“default”, “fr”})
  - GreetingService primaryGreetingService(GreetingServiceFactory greetingServiceFactory) {
    - return greetingServiceFactory.createGreetingService(lang, “fr”); }
- }

- Cleaner Configuration for more complexes classes where you can put the configuration all in one file instead of spread all over.

Spring Boot Configuration

- SpringBoot has been doing a lot of the configuration under the covers for use.
- Dependency Management
  - Maven or Gradle are supported for curated (to pull together, sift through, and select for presentation) dependencies
    - Give you a set of dependencies that are going to work out of the box with that Spring Boot Version.
  - Each version of Spring Boot is configured to work withy a specific version of the Spring Framework
    - Overriding the Spring Framework Version is not recommended
- Maven Support
  - Maven projects inherit from a Spring Boot Parent POM
    - When possible do not specify versions in your POM. Allow the versions to inherit from the parent
      - Specifying the version could cause weird conflicts, so whenever possible don’t
      - Been specified tow work with the library versions
  - The Spring Boot Maven Plugin allows for packaging the executable jar
- Gradle Support
  - Depends on Spring Boot Gradle Plugin and requires Gradle 3.4 and later
    - Supported of curated dependencies, packaging as jar or war and allow the application to be run from the command line.

Spring Boot Configuration

- Spring Boot Starters
  - Starters are top level dependencies for popular Java Libraries (plugins)
  - Will bring in dependencies for the project and related Spring components
    - Example Starter ‘spring-boot-starter-data-jpa’ brings in hibernate, Spring Data JPA and related Spring dependencies
- Spring Boot Annotation
  - Main annotations to use which include
    - @Configuration — Declare class as Spring Configuration
    - @EnableAutoConfiguration — Enables auto configuration which makes a lot of decisions about components being used (eg db)
    - @ComponentScan — Scans for components in current package and all child packages.
- Disabling Auto Config
  - Add a lot of configuration classes in sullied Spring Boot Jar
  - Exclude classes @EnableAutoConfiguration(exclude={DataSourceAutoConfiguration.class})

Spring Boot Configuration Demo

- Open the annotation to see the source code and the @EnableAutoConfigurtion, @ComponentScan will be visible
- mvn spring-boot:help -detail shows all the options and descriptions about them
- mvn spring-boot:run -Drun.arguments—debug – In the output produces information about classes that did
- For Intellij Edit Configurations, we can see the options that were configured.
- Sprng Boot is a wrapper around spring not a web application

Spring Boot

- Spring Boot
  - A faster and more understandable initial experience with the Spring Web Framework
  - Provide families of features with the minimal configuration
  - Does not generate code
  - Dynamically wire up beans
  - Favors used provided defaults when found
  - Spring Boot Starters throw maven and gradle
    - Help to reduce the number and manually added dependencies by adding one dependency
    - <parent>
      - <groupdId>org.springframework.boot</groupId>
      - <artifactId>spring-boot-starter-parernt</groupId>
    - </parent>
    - <dependency>
      - <groupId>org.springframework.boot</groupId>
      - <artifactId>spring=boot-start-web<artifactId>
    - <dependency>
    - No code , but pull in other dependencies

Spring Boot

- Project Structure is important
  - Application Class Spring Boot will auto scan for Spring Managed classes in this packages and its sub packaged to create the WebApplicationContext. with SpringMVC and Sprng Managed Bean
  - CSS and JavaScript will be in the src/main/resources/static directory
  - Templates will contain the HTML files using thyme leaf
  - Configuration – Key values pairs placed there are read into Spring Environment Object that configures the applications.
    - Could be a YAML File
  - Generated basic test class using @SpringBootTest a means to inflate test and tear down WebApplicationContext
  - pom.xml
- Executable Jar
  - included support for embedded Tomcat, Jetty and Undertow servers by using the correct POM
  - Uses a new type of Application for Embedded Servlet Support
  - EmbeddedWebApplicationContext – A special type of WebApplicationContext
    - bootstraps itself by searching for a single EmbeddedServletContainerFactory – Ex. TomcatEmbeddedServetContainerFactory
- Can be create as war or executables jar (better with docker or kubernetes since it contains everything
  -

Spring Bean Scope.

- Spring Bean Scopes
  - scope can be a prototype – prototype
  - Singleton (default) : Only one instance of the bean created in the IoC container
    - It is based off the signaler instance per id attribute of the BeanDefinition
  - Prototype – New Instance is created each time the bean is requested
  - Request – A single instance per http request.
    - Only valid in the context of a web-aware Spring Application Context
  - Session – A single instance per http session
    - Only valid in the context of a web awaere Spring Application Context
  - Global Session – A single instance per global session. Typically used in a Portlet context.
    - Only valid in the context of a web aware Spring Application context
  - Application – Bean is scope to the lifecycle of a Service Context.
    - Only valid in the context of a web aware
  - Websocket – Scope a single bean definition to the lifecycle of a WebSocket.
    - Only validin the context of a web-aware Spring Application Context
- Custom Scopes
  - Spring Scopes are extensible and can define your own scope by implementing Spring's Scope Interface, but cannot override the Signleton/Prototype

Spring Bean Scope

- Declaring Bean Scope
  - No declaration needed for singleton scope
  - In Java configuration use @Scope annotation
  - In Xml Configuration scope is an xml attribute of the bean tag

Spring Pet Clinic – Load Data on Startup with Java
Implement Spring configuration
List Owners
List Vets
Spring Pet Clinic – Auto Generate Map IDs

- Standard : For Startup Code (data or functions) put them in a package that ends with bootstrap.
- Spring Boot
  - Interface Command Line Runner with run which takes arguemnts into the environment
- Implement Spring configuration
- List Owners
  - <!-- .&@thymesVar id=”owner” type=”guru.springframework.sfgpetclinic.model.Owner”*/-→ // hint for Intellij intellisense
  - Thymeleaf : – loop through a row
    - <tr:each=”owner : ${owners}”> // knows owner by comment statement above.
      - <td th:text={owner?.id}</td>
    - </tr>
- List Vets
  - With the model object we can add an attribute that contains our data example. “vets” will contain the list of vets
- Spring Pet Clinic – Auto Generate Map IDs
  - Good use of exception. You can pass in the map and if null then set the index to one and then create the object using lazy valuation.

Property Source

- Introductions – What changes from environments you want to externalized (ex. From dev to test)
- In the resources property files are stored – example name datasource.properties
- ${} is the Spring's expression language
- PropertyPlaceHolderconfigurator
- To use the @Value we must have the annotation @Bean, @Inject etc..
- Example
  - @Configuration
  - @PropertySource("classpath:datasource.properties")
  - Public class PropertyConfig {
    - @Value("${guru.user}") // The PropertySourcePlaceHolderConfigurer will match the @Value with the declaration beneath
    - String user;
    - @Value("${guru.password}") // The expression will go into the Spring Context and get that value from the externalized property
    - String password;
    - @Bean
    - Public static PropertySourcePLaceHolderConfigurer() {
      - // Will Scan and read the file
      - PropertySourcesPlaceHOlderConfigurer configuration = new PropertySourcePlaceHolderCofnigurer();
      - return configuration;
    - } }
- Can Have @Autowired public DepartmentService(@Value("Classics") String name)

Spring Environment Properties Multiple Property File

- Intellij : Edit Configurations → Edit Environment Variables
- Can autowire the environment variable
  - @Autowired
  - Environment env;
  - Can use env.getProperty.getProperty(“userName”);
- The environment variables can override the variables in the property files.
  - Good use password can be set as an environment
- Multiple Property Files
  - @PropertySources{ (@PropertySource(“classpath:file1”), (@PropertySource(“classpath:file2”) } instead of @PropertySource after the @Component from the previous example.
  - Common to have different property files for each resource (database, jms etc..) so that you can mix and match as needed;
  - Should have unique properties since they could overwrite each other without complaining. (Think namespaces)

EL and SPL

- PropertySourcePlaceHolderConfigurer can resolve place holder expression using @Vallue annotations against the current Spring Env.
  - Note the static since it must be resolved before any other bean as it is amending meta data
- Source can be set a @PropertySource
- Example
  - Service getService(@Value({“$department”}, int department) {
    - return new EmployeeServiceImpl(new EmployeeDaoImpl(), department);
  - }
  -
  - @Value(“${department}”) private int department
- Could Inject the values from property files into our configuration classes in order to keep our domain classes annotation free
- EL is a literal replacement, but Spring EL can execute code
  - @Value(“#{ T(java.lang.Math).random() * 100.0 } “)
  - private double discount
  - <bean id=”shop” class=”$(shop.class)”> <property name=”discount” value=”#{ T(java.lang.Math).random() * 100.0}” /> </bean>
  -

EL and SPL

- Can locate Spring Managed Bean
  - #{id}, #{id.property}, #{id.method|
  - #{employeeService} – get the bean with an id of EmployeeService
  - #{employeeService.employees} – Translates to executing the method getEmployees() via reflection
  - #{employeeService.allEmployees()) – does not suffix get, but executes allEmployees
  - @Value(“#(employeeDao”)} gets the bean with an id of “employeeDao”
  - @Vaue(“#employeeService.dao.getOne(1L)”})
    - Execute the methods on a bean whose id is employeeService.getDao().getOne(1L)
- SPEL Selectors
  - Can filter collections
    - We can use SPEL to filter this map into another Collection such as a list of employee in a department
    - Example
      - private static Map>Integer, Employee> amp;
      - public Map<Integer, Employee> getMap() { return map; }
      - @Value(“#employeeDaoImp.map.values().?[department == 7L]}”) // ? (filter), ^ first matching, $ last matching
      - private List<Employee? employeesInDepartment; // Where the filtered values goes
      - public List<Employee> getEmployeeeInDepartment2() { return employees inDepartment2; |

EL and SPL

In addition to returning all the selected elements, it is possible to retrieve just the first or the last value. To obtain the first entry matching the selection, the syntax is ^[...] while to obtain the last matching selection, the syntax is $[...].

Combine EL and SPEL and getting and getting a single property

```
@Value("#{employeeDaoImpl.map.values().^[department == 7L]}")
private Employee  empl;
@Value("#{employeeDaoImpl.map.values().$[department == 7L]}")
private Employee  emplLast;
```

```
@Value("#{employeeDaoImpl.map.values().$[department == '${dept}'].firstName}")
private String emplName;
public String getEmplName() {
  return emplName;
}
```

SPEL has systemProperties – @Value(“#{systemProperties[‘os.name’]}”)
private String str;

EL and SPL

- Junit
  - We can set our own System Properties by using ApplicationContextInitializer to setup value for use in the appliationContext
  - Define the initializer and register it with your Junit
  -

```
public class MyInitializer implements ApplicationContextInitializer<ConfigurableApplicationContext> {
@Override
public void initialize(ConfigurableApplicationContext applicationContext) {
     String str = System.getProperty("os.name").toLowerCase()
                  .startsWith("windows")? "hello windows": "hello other";
     System.setProperty("myproperty", str);
}
}
```

```
@ExtendWith(SpringExtension.class)
@ContextConfiguration(classes=ApplicationConfig.class, initializers =
MyInitializer.class)
public class StudentServiceTest
```

```
@Value("#{systemProperties['myproperty']}")
private String str;
```

```
@Value("#{systemProperties['os.name'].toLowerCase().startsWith('windows')?
'hello windows' : 'hello other'}")
private String str;
```

Conditionals

- Anywhere we define a Spring Bean, we can optionally add a condition
  - Only if the condition is satisfied will the bean be added
  - Can be applied at the @Bean, @Component
- Spring has defined @Conditional on
  - @conditionalOnProperty annotation allows us to load bean conditionally on a certain environment proeprty
  -

```
@Bean(name="employeeService")
@ConditionalOnProperty(
    value="env",
    havingValue = "test",
    matchIfMissing = true
)
EmployeeService service(){
    EmployeeServiceImpl service =
         new EmployeeEmployeeImpl();
    return new
    EmployeeServiceProxy(service);
}
```

```
@Bean(name="employeeService")
@ConditionalOnExpression(
  "'${env}' == 'test'"
)
EmployeeService service(){
    EmployeeServiceImpl service =
         new EmployeeServiceImpl();
    return new EmployeeServiceProxy(service);
}
```

@ConditionalOnExpression allows us to load bean conditionally on certain expression being true

Conditionals

- @ConditionalOnBean – allows us to load beans conditionally
  - Ex. The EmployeeService is only if there is a bean of class EmployeeDao in the Application Context
- @ConditionOnResource – Load a bean depending if a certain resource is present
  - Conditionals can be combined – Everything must resolved to true or create compound expression using &&
- Allows use to have bean with the Same Names, but can load the bean for the right conditions
- Can create our own Condition class only one function matches

```
@Bean(name="employeeService")
@ConditionalOnBean(
  EmployeeDao.class
)
EmployeeService service(){
    return new EmployeeServiceImpl();
}
```

```
@Value("${department}")
private int dept;
@Bean(name="employeeService")
@ConditionalOnResource(resources="/application.properties")
EmployeeService service(){
      EmployeeServiceImpl service =  new EmployeeServiceImpl();
      service.setDepartment(dept);
      return service;
}
```

Spring Boot Application Properties

- The previous slides for external configuration were legacy examples
- application.properties is the default property file.
  - Automatically brought in by SpringBoot
- In the Resource Directory create an application.properties
  - Can add in your own properties
  - Keep the @Value with the declaration bean to match the property name with the class’s property name.

Introduction to YAML

- More suited for describing data structures automatically supports child/parent relationship
- Spacing is important 2 spaces and space between the colon and the value
- Example – under resources example.yml
  - # this is a comment
  - name: John // Create a property
  - Names: # Comment
  - - John #Comment
  - - Paul
  - pound_sign: “#” // Make the pound a value instead of syntax
  - Book: // Creates an object
    - author: Joe Buck
    - Publisher: random house
  - truth: Yes // Boolean can also be True or true, false can be no
  - string_val: “This is it’s quote”
  - Include_new_lines: |+ # Keeps the new lines , >+replace new lines with spaces (+ keeps all ne lines, - no newlines, > only single new line only
    - Asdf
    - adfewf

Spring Boot YAML Properties

- The file extension is yml and Spring Boot will look for application
- Example
  - guru:
    - jms:
      - username: JMS Username
      - password: somepass
      - url: SomeUrl
- Can have application.properties and application.yml at the same time
  - The YAML file can use properties in the application properties anyway.
- Spring Boot is reading all the properties files
- Example Ports
  - server.port
  - server.servlet.context-path=/work
- The values in application.properties is found in org.springframework.core.env.Environment object
-

Spring Boot YAML Properties / Properties

- @Inject
- private Environment environment
- @GetMapping(“/test”)
- pubic String getHome() {
  - String title = environment.getProperty(“salutation”);
  - String javaversion = environment.getPorperty(“java.runtime.version”);
  - }

Property Hierarchy Used by Spring Boot

- Highest Priority Devtools global settings properties
- Lowest Default Properties (SpringApplication.setDefaultProperties) is overwritten by anything
- When we have the option for *.proprieties or *.yml they are treated equally
- https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-external-config.html for the treachery of setting parameters
  - The baseline development will probably be inside your jar
-

Spring Boot / Configuration Classes

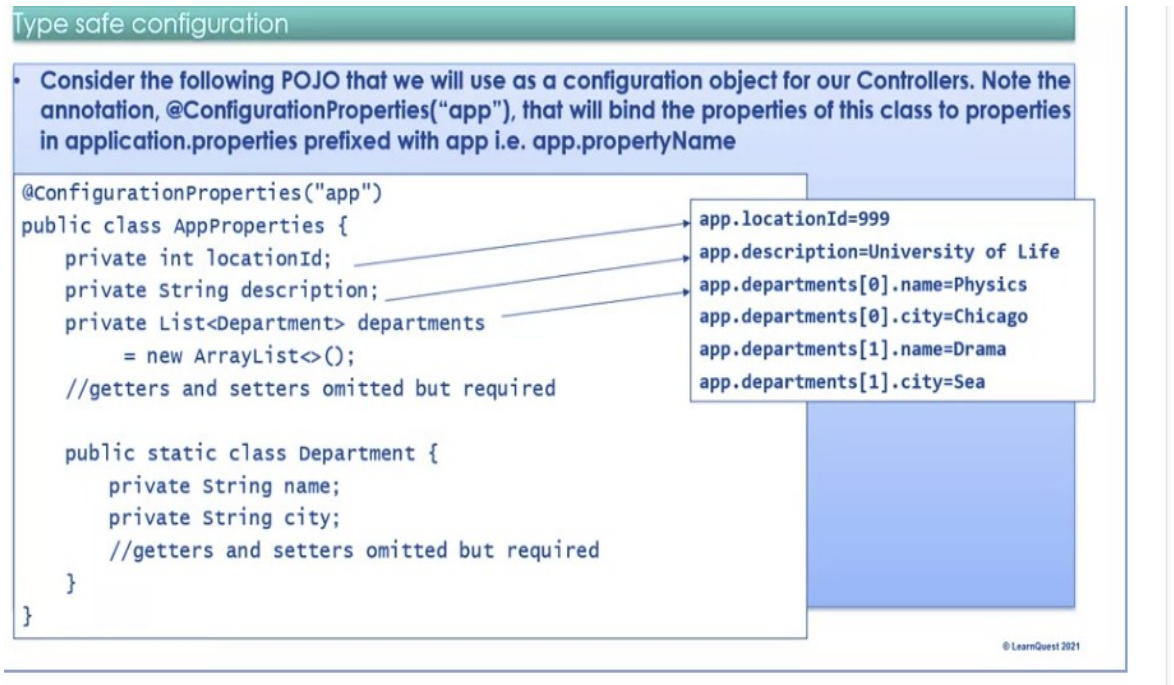
- We have been using @Value and the type conversions are accomplished validation are not done when they are loaded
- Spring Boot provides an alternative method of working with properties and validate the configuration of your application
- Preferred since they are type safe can be validate by JSR2020 validators

@ConfigurationProperties tells them to look for app properties

execute the setters for setting configuration. Provides use with type checking

Need to provide @EnableConfigurationProperties(value=AppProperties.class) to enable

```
@Inject private AppProperties appProperties
@GetMapping
public String get() {
        return appPrperites.getLocationId();
}
```



Bean Definitions in the Applications

- @SpringBootApplication contains @Configuration
- In the Classname annotated with SpringBootApplication it can use the @Bean

Spring Boot Setting Properties with Profiles

- SpringBoot allow you to work in the profile name into the filename
- Can set the profile name inside the yaml file
- The properties for profiles will take precedence for properties with a profile
- A powerful tools to manage different environments with Spring Boot
- Spring has a property
  - sprint.application.active=de for properties file
  - YAML
    - Add the original YAML from the previous example above
    - --- 3 dashes represent a file separator
    - spring:
      - profiles: de // The Spring Profile will override the above
    - guru:
      - jms:
        - username: JMS Username $$$$ German
- With the help of Spring Boot then Spring will store everything into the context automatically

Thymeleaf

- A Java Template Engine that is not a web framework under the Apache 2.0 Open Source Licence
- Produces XML, XHTML and HTML5 and is a replacement for JSP
- A Natural template engine meaning the templates are viewable in a web browser
- Not tied to the environment
  - Example can be used for producing emails
  - Don’t need a servlet engine or any type of web environment
- Thymeleaf vs JSP
  - Valid HTML documents you view in the browser
    - JSP are not valid HTML and look awful in the browser
  - The Natural templating ability allows you to perform rapid development without the need to run a container to parse the template/JSP to view the product in the borwser which speeds development time (don’t needTomcat and refreshing
- Extends the namespace with so that the HTML is valid. The Tags are compliant and are ignored by the browser
  - JSP has a lot of tags that the browser does not know what to do with

Thymeleaf

- Performance
  - Benchmarks slower than other template engines such as JSP and Velocity (Spring 5 no longer supports)
    - 3.0 did bring significant performance improvements
- Notes on template
  - Declaring an xml namespace : https://www.thymeleaf.org
  - Th:href="@/webjars/bootstrap/3.3.7-1/css/bootstrap.min.css"
    - Run directly in browser → The browser will show the css from the cdn
    - Run in springboot the tag will be replace with a reference to the web jar

Create Index Page

- Example – A control where the context page is blank, / or /index and return the index as the page name
  - @Controller
  - Public class Index Controller
    - @RequestMapping({“”, “/”, “/index”})
    - Public String getIndexPage() {
      - Return index
      - }
- Example index.html
  - <!DOCTYPE html >
    - <html lang=”en” xmlns:http://www.thymeleaf.org”>
    - <head>
      - <meta charset=”UTF-8” ‘/>
      - <title>Recipe Home</title>
    - </head>
    - <body>
      - <h1>My Recipes</hi1>
    - </body>

HTTP Request Methods

- Started as a telnet friendly protocol
  - Use ssh instead of telnet
- HTTP History : Version 1.0 from 1991 to 1995
  - Contain request line with HTTP Version number followed by headers such as User Agent
  - Response status followed by header such as Content-type, Content-length, Expires etc..
- Http Standard were developed by IETF and W3C designed a design document
- HTTP/1.1. release in 1997
  - Added support for keep alive connection, chunked encoding transfers, byte range request, transfer encoding and request pipelining
  - Added later : Request Added encoding, charset, cookies and response added encoding, charset and cookies
- HTTP 2.0 Standardized in 2015 (Performance release)
  - Transport performance was a focus
  - Improved load speed by lower latency and higher throughput
    - Helps in scaling applications.

HTTP Request Methods

- Request Methods (verbs) indicate the desired action to be performed.
- Get
  - Request for a resource (html file, javascript, file image)
  - Used when you visit a website
- Head
  - Ask for meta information without the body
- Post
  - Used to post data to the server (insert new data)
    - An example would be a checkout form
    - Post is a create request
- Put
  - Is a request for the enclosed entity be stored at the supplied URI. If the entity exist it is expected to be updated
  - Put is a create or update request
-

HTTP Request Methods

- Delete
  - A request to delete the specified resource
  - Not supported by the HTML Standard.
- Trace
  - Will echo the received request
  - Can be used to see if the request was altered by intermediate server
  - Example : Proxy Servers can change the request along the way
- Options
  - Returns the HTTP Methods support by the server for the specified URL
- Connect
  - Converts the request to a transparent TCP/IP tunnel typically fro HTTPS through an unencrypted HTTP Proxy
  - The author has never seen a direct use for
- Patch
  - Applies partial modifications to the specified resource

HTTP Request Methods

- Safe Methods
  - Considered safe to use because they only fetch information
  - Do not cause changes to the server unless you are returning dynamic data
  - Get, Head Options and Trace are all Safe Methods
- Idempotent Methods
  - A quality of an action such that repetitions of the action have no further effect on the outcome
  - put and Delete are Idempotent Methods
  - Safe Methods (Get, Head, Trace, Options) are Idempotent
  - Being truly idempotent is not enforced by the protocol, but part of the standard
- Non-Idempotent Methods
  - Post is not Idempotent
  - Multiple post are likely to create multiple resource
    - Example. Ever seen websites asking you to click submit only once
- Cacheable : Get, Head, Post, Patch
- Non Cacheable : Put, Delete, Connect Options, Trace

HTTP Status Code

- Status Codes
  - 100 Informational
  - 200 Indicates successful request
  - 300 series are redirections Example a website has moved
  - 400 client errors
  - 500 server side errors
- Common HTTP Status Codes
  - 200 Okay
  - 201 Created
  - 204 Accepted
  - 301 Move Permanently Will get the new location back
  - 400 Bad Request Example a media type that the media does not accept
  - 401 Not Authorized
  - 404 Not Found
  - 500 Internal Server Error
  - 503 Service Unavailable

Chrome Developer Tools
FireBug
FireFox Developer Edition
Safari Web Inspector
Axis TCPMon

- Chrome Developer Tools
- FireBug – Changing to Firefox Developer Edition
- Firefox Developer Edition
  - Menu Item : Tools → Web Developer
- Safari
  - Has a Storage Tab to show the local storage
- Axis TCPMon – Has plugins for all IDE
  - Can Listen on Port and pass traffic to another port
  - Can see the request sent back and forth.

Spring Developer Tools

- Added to project via artifact ‘spring-boot-devtools’
- Developer tools are automatically disabled when running a packaged application (java -jar)
- By default, not include in repackaged archives
- Features
  - Automatic Restart
    - Triggers a restart of the Spring Context when classes change
    - Use two classloaders : One for your application and one for project jar dependencies
    - Restarts are very fast since only your project classses are being loaded.
    - Eclipse
      - Restart is triggered with save (which by default will compile the clas which triggers the restart)
    - Intellij
      - By default you need to select Build / Make Project
      - Advanced setting you can change to make this more seamless (Automatic Saves)
  -

Spring Developer Tools

- Template Caching
  - By Default templates are cached for performance
  - But Caching will require a container to restart to refresh the service
  - Developer tools will disable template caching so the restart is not required to see changes
- Live Reload
  - Automatically trigger a browser refresh when resources are changed
  - Spring Boot Developer Tools includes a LiveReload server
  - Browser plugins are available for free download at livereload.com

Intellij Compiler Configuration for Spring Boot Development Tools

- How to compile the files on save which will trigger Spring Tools to load the files again
- Access the registry : Command Shift A and search Registry
  - Want to find the key compiler.automake.allow.when.app.is.running
- Now Project Setting → Compiler → Build Project Automatically
- Stop and Restart Intellij

Spring Pet Clinic – Static Resources
Spring Pet Clinic – Copy Master Template from Spring Pet Clinic

- Spring Boot defines a Static folder that has a resource which has fonts and pictures.
- Git does not publish empty folder which can be done with an option
- Spring Pet Clink – Copy Master Template from Spring Pet Clinic
  - Master Template – A file with
    - html xmlns:th="http://www.thymeleaf.org" th:replace"~{arguments/layout :: layout (~{::body}, 'home')}" >
- There is a fragments
  - <th:block th:include="${template}" /> Injects the welecome page insert here. In this file is the header and navigation (reuse)

Spring Pet Clinic – Implement Web Resource Optimizer for Java

- Springboot does not process less resources
- Using wro4j → A project to improve your web application page loading time.
  - Helps to keep static resources (CSS, JS) well organized, merged and minify them at run time using a simple filter or build-time
  - Maven has a dependency webjars
    - Web jars → A jar with the web resources
- Dependencies
  - Webjars-locator-core
  - Jquery
  - boostrap
- Build plugin which run wro4j phase is generate resource
  - Configuration
    - Java Class
    - Css Destination folder
    - WRO Files – The XML File
    - Extra Config Files → The properties files preprocess and post processor
    - The Context Folder → Less Directory

Spring Pet Clinic – Implement Web Resource Optimizer for Java

- Since it is a maven plugin run mvn clean and mvn package and you will it compile the less
- We are using wro4j plugin to generate resource from less files and combine those.
-

Spring Pet Clinic Apply Master Layout to Index Page

- Find the welcome message for english
  - <h2 th:text="#{welcome}">Welcome</h2>
- Replacing the th:replace="~(fragments/layout :: layout (~(:body), 'home)}, 'home')">

Spring Pet Clinic – Internationalization Properties
Apply Master Layout to Owner page
Apply Master Layout to Vet Page

- Under the Resource Directories we have the Messages folder which contains the property names.
  - Example : message_de.properties
  - Example : message_en.properties — This file is empty and will go to the default message.proeprties
  - Example : message.properties — Overall bottom of the line default
  - If there a file message_en_us.properties — This would be the files since the text in the thyme file is message_en_us
- In Application spring.messages.basename = messages/messages → provide the location of the messages where the first message is the directory and the second message is the resource bundle
- Apply Master Layout to Owner Page
  - th:replace="~{fragments/layout :: layout (~::body}, 'owners'})"> add to the HTML Tag
    - Tell thymeleaf to apply the common layout template
- Apply Master Layout to Vet Page
  - th:replace="~{fragments/layout :: layout (~::body}, 'vets'})"> add to the HTML Tag
- If files are missing something might have happened to the target directory. Run the maven package again.

Spring Pet Clinic – Create Pet Type, Pet , Visit Entities
Create Pet Type, Pet and Visit Entities
Add Contact Info to Owner
Create Pet Type Map Service

- Currently creating and modifying the POJO(s) to contain the data.
- Create Pet Type, Pet and Visit Entities
  - Just an Object Speciality that contains a String
  - In the Vets we have Vets<Specialty> specialities
- Add Contact into to Owner
  - Creating an owner which extends Person
- A pro tip : Just remove all the Setter and Getters and then regenerate them to keep them in order
- Create Pet Type Map Service
  - Some of the thee we have already created and we have findById, save, deleteById, delete, getNextId
  - extends CrudService<PetType, Long>
  - Add the @Service : These are POJO, but perform operations

Spring Pet Clinic – Pet Type Data on Startup
Enhance Owners and Pets with Contact Info on Startup
Create Speciality Map Service
Spring Pet Clinic – Add Specialities to Vets on Startup

- For the Data Loader class that we have create we are adding a PetDoc and a PetType Service that Spring will autowire in
- When we are loading the data we save the PetType(dog). We are establishing the PetType for later use.
- Enhance Owners and Pets with Contact Info
  - We will need the two owner with Name and Address and Telephone
  - Enhance the save in two way
    - If a null object is passed in then throw a null
    - For each owner check all Pets
      - does not have a PetType throw a runtime error. Next check type PetType and the Id is null then save it
      - Check the Pet ID and if the id is null then save it
  - Now we create a new Pet (Type, Owner, BrithDate, setName) and store it in the Owner
- Create Speciality Map Service
  - Creating a Specialities which extends CrudService<Speciality, Long> which has the @Service
  - Take a look at how Intellj infers Generics there may be some work there for you.
- Add Specialties to Vets on Startup – Move the data initialization into its own function and call loadData() if the size of the data structure is 0
- Interesting you can have a private final <Type> instance and set in the constructor

Spring Pet Clinic – Fix Broken Links

- Not Implemented Handling
  - Creating a new template (Thymeleaf template) by adding http://www.thymeleaf.org namespace call notImplemented.html
  - @RequestMapping(“/find”)
  - Public String findOwners() { Return notImplemented }
- If you have a class that has controller annotation and does not define a request mapping then the @Request Mapping completed from the root
-

Spring and Data Access

- Developers usually develop applications when developing for Spring use embedded Databases
  - Can Autoconfigure H2, HSQ, Derby – Need to simply include a build dependency
- JPA is a specification not an implication – It provide a set of interfaces and methods for interacting with a variety of ORMS
- JPA abstracts the specifics of an ORM framework such as Hibernate
  - Instead of session use Entity Manager
  - Hibernate can be used as a Implementation Provider
- Database Configuration
  - application.properties – spring.datasource.url, spring.datasource.username, spring.datasource.password, spring.datasource.driver-class-name
  - org.apache.derby.derbyclient.10.14.2.0
- Connection Pooling
  - called OnDemand
  - Uses spring-boot-starter-jdbc by default uses tomcat-jdbc conneciton pool
    - Use dbcp2 – remove the dependency and add commons-dbcp2
  - application properties for dbcp2 spring.datasource.dbcp2.initial_size=0, spring.datasource.dbcp2.max-idle, spring.datasource.dbcp2.min-idle, max-total

Spring and JPA

- Spring Boot Abstractions
  - Must have two key files in the classpath: schema.sql, data.sql containing sql statement to insert those tables
  - Using Hibernate Over JPA – Need to tell SpringBoot not to generate a schema using the POJO classes : spring.jpa.hibernate.ddl-auto = none
- Transactions open and close EntityManager that enclose Hibernate sessions
  - lazy loading – save resources by not loading related objects into memory when we load the main object
  - Hibernate use wrappers and proxies to implement lazy loading
  - lazy loaded data has two steps – load the main object then retrieve the data within its proxies
    - Hibernate must have an open session
    - If the second steps happens after the transaction has closed – LazyInitialization Exception
      - enable_lazy_load_no_trans property each fetch of a lazy entity will open a temporary session and run inside a separate transaction
        - spring.jpa.properties.enable_laby_load_no_trans=true
    -

Spring and JPA

- Spring Boot Abstractions
  - Must have two key files in the classpath: schema.sql, data.sql containing sql statement to insert those tables
  - Using Hibernate Over JPA – Need to tell SpringBoot not to generate a schema using the POJO classes : spring.jpa.hibernate.ddl-auto = none
- Transactions open and close EntityManager that enclose Hibernate sessions
  - lazy loading – save resources by not loading related objects into memory when we load the main object
  - Hibernate use wrappers and proxies to implement lazy loading
  - lazy loaded data has two steps – load the main object then retrieve the data within its proxies
    - Hibernate must have an open session
    - If the second steps happens after the transaction has closed – LazyInitialization Exception
      - enable_lazy_load_no_trans property each fetch of a lazy entity will open a temporary session and run inside a separate transaciton.
        - spring.jpa.properties.enable_laby_load_no_trans=true
    -

Spring and JPA – Entity Managers

- Major classes
  - A connection to the data is representative by an Entity Manager instance which performs operations on a database
  - PersistenceContext – Where managed entity instances are cached
  - An Entity Manager is constructed for a specific database
- Can also use JDBC Template Classes , integrating JPA and Spring are better solutions

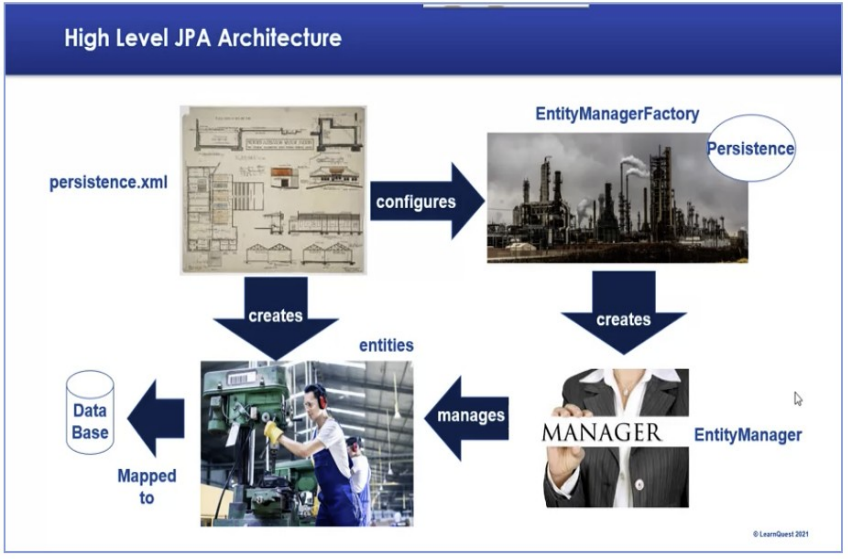
```
@Entity
@Table(name = "EMPLOYEE")
public class Employee {
    @Id
    @GeneratedValue(strategy=GenerationType.IDENTITY)
    @Column(name = "EMPID")
    private long id;
    @Column(name = "FIRSTNAME")
    private String firstName;
    @Column(name = "LASTNAME")
    private String lastName;
    @Column(name = "DEPTID")
    private long dept;
    public Employee() {
      super();
    }
```

Must have @Entity and if table name does not match class name use the @Table

Must have a unique identifier that reflects primary key in associated table

Map properties to Table Columns. Can be placed above a setter

Must have a default constructor, can have other overloaded constructors. Bean should be a Java Bean.



Spring and JPA – Entity Managers

- A transactionally (Managed) Scoped Context
  - Example
    - @Named
    - public classs EmployeeDaoImpl implements EmployeeDao {
      - @PersistenceContext
      - private EntityManager em;
      -
      - public Employee getById(long id) { return em.find(Employee.class,id);
  - Our Dao is a plain POJO.
    - Spring recognized the @PersistenceContext annotation and injects an EntityManager with its PersistenceContext created by the EntityManagerFactory
      - Transaction scoped by default
    - if the Persistence Context is already present then it will associated the persistence context with the Injected EM. If not then create a new on
- @PerisstenceContext – Injects the EntityManager a transactionally scoped persistence
- @PersistenceUnit injects the EntityManagerFactory where you define the scope of any transaction programmatically

Spring and JPA – Entity Managers

- Extended Persistence Context
  - @Named
  - public class EmployeeDaoImpl implements EmployeeDao {
    - @PersistenceUnit private EntityManagerFactory emf;
    - @PostConstruct
    - private void initi() { em = emf.createEntityManager(); }
    - private EnitityManager em;
    - public Employee getById(long id) { return em.find(Employee.class, id); }
  - Sometimes you want a persistence context to last longer than the life of transaction. Create the EntityManager yourself and inject
    - If you create it you must autoClose it.
- Crud Operations
  - public interface EmployeeDao {
    - Employee getId(long id);
    - @Transactional long add(Employee employee) .// @Transactional – Run in an Transaction (Transaction scope)
    - @Transactional Employee update(long id, long department);
    - @Transactional void delete(long id) }

Spring and JPA – Entity Managers

- Implementation of the interface

```
@PersistenceContext private EntityManager em;
@Override
public Employee getByID(long id) {
    return em.find(Employee.class, id);
}
@Override
public long add(Employee employee) {
    em.persist(employee); //returns void
    return employee.getId();
}
@Override
public Employee update(long id, long department) {
    Employee employee = em.find(Employee.class, id);
    employee.setDept(department);
    return employee;
}
@Override
public void delete(long id) {
    Employee employee = em.find(Employee.class, id);
    em.remove(employee);
}
```

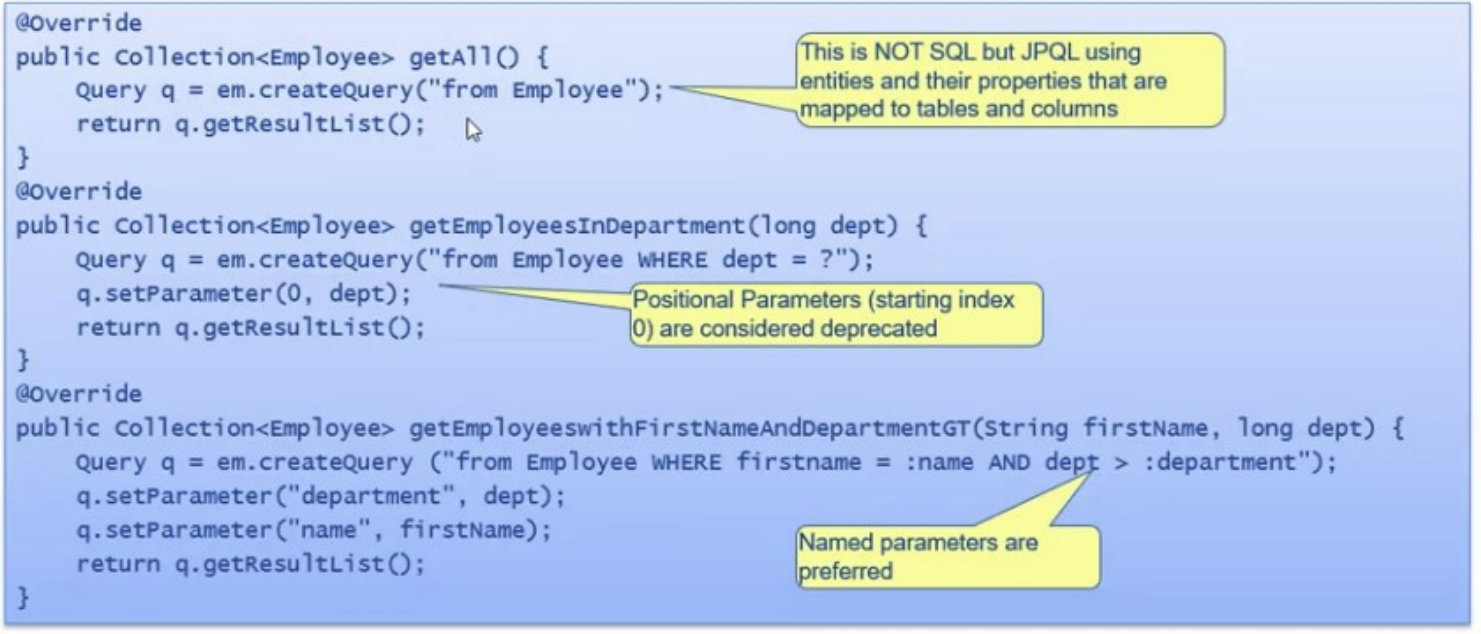
Since this method runs in a transaction, once it completes, the transaction commits and flushes the state of a bean from the EntityManager's cache to the database

If a transaction is provided it will be used

When you finish the method flush the context of the database done by the abstractions

Spring JPA – Queries

- Create Queries from employees
-



Spring Data – TypeQuery

- TypedQuery Interfaces
  - Extends the Query interface when the Query ResultType is unknown or when a query can return polymorphic results
  - Usually use the TypedQuery Interface
  - Usually easer to iterate over the returned Collection
-

```
public Collection<Employee> getEmployeeswithFirstNameAndDepartmentGT(String firstName,
long dept) {
    TypedQuery<Employee> q = em.createQuery ("from Employee WHERE firstname = :name AND
    dept > :department", Employee.class);
    q.setParameter("department", dept);
    q.setParameter("name", firstName);
    return q.getResultList();
}
```

Spring Data – Named Query

- Trigger by name

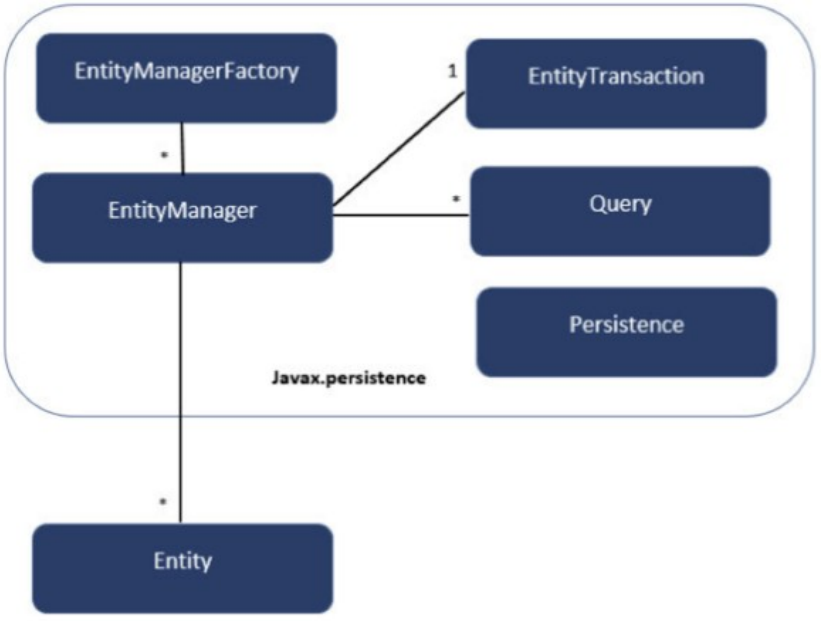
The Entity is annotated directly with the Query

```
@Entity
@Table(name = "EMPLOYEE")
@NamedQuery(name="Employee.findByFirstName",query="Select emp " +
 "from Employee emp where emp.firstName LIKE :name")
public class Employee {
```

```
public Collection<Employee> getEmployeesWithFirstNameLike(String name) {
    Query q = em.createNamedQuery("Employee.findByFirstName") ;
    q.setParameter("name", name);
    return q.getResultList();
}
```

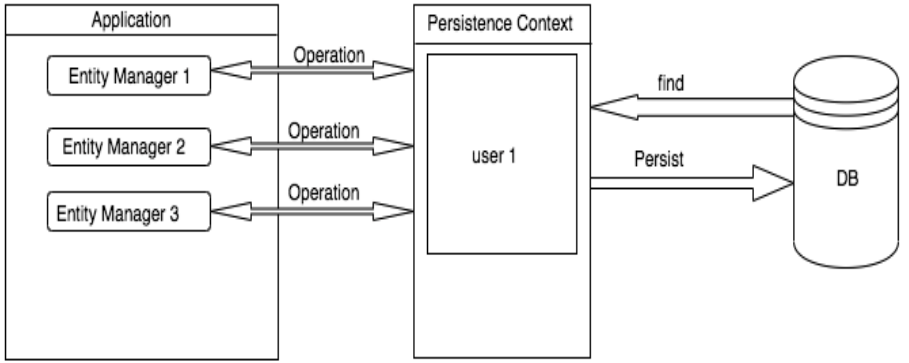
Spring JPA – Architecture

- javax.persistence
  - EntityManagerFactory
  - EntityManager — Manages persistence operations on objects
  - EntityT ransaciton — One to One relationship with EntityManager
  - Query
  - Persistence — static methods to obtain EntityManagerFactory
- Persistence Context
  - First level cache where all the entities are fetched from the database or save to the database
  - Keep track of changes made to the entity
  - EntityManager allows use to interface with the database



Spring JPA – Architecture

- Transaction Scope Persistence Context
  - As soon as the transaction finishes the entities in the persistence context are flushed
  - When we perform an operation if there is no persistence context it will be created
  - default persistence context type is PersistenceContextType.TRANSACTION (@PersistenceContext)
- Extended Scoped Persistence Context
  - Can space across multiple transaction.
  - Can persist the entity with the transaction, but cannot flush it without a transaction
- Need to use @PrsistenceContext(type=PersistenceContextType .Extended)
- In the Stateless session bean the extended persistence context is one component is completely unaware of any persistence context of another component
- Example
  - Let's say we persist some entity in a method of Component A, which is running in a transaction.
  - We then call some method of Component B.
  - In Component B's method persistence context, we will not find the entity we persisted previously in the method of Component A.
- See Example at bottom ==https://www.baeldung.com/jpa-hibernate-persistence-context



Transaction Persistence Context

JPA Entity Relationships

- Type of Relationships
  - One to One
  - One To Many
    - Typically a Collection (List, Set, Map)
  - Many to One
  - Many to Many
    - Each list has a List or Set reference to the other
    - A join table is used to define the relationships
- Unidirectional vs Bidirectional
  - Mapping is done one way. One side of the relationship will not know about the other
  - Both side know about each other.
    - Generally recommended to use Bidirectional since you can navigate the graph in either direction
  - Performance there is no different
- Model the Primary Foreign Key Relationships

JPA Entity Relationships

- Owning Side
  - The Owning side in the relationship will be table with the foreign key in the data
  - One to Many – The Owning side usually defined on the many side of the Relationship. Side which owns the foreign key
    - 2 tables class / students – That would be the students table which owns the foreign key found in the class table
  - The @joinColumn annotation defines the actual physical mapping on the owning
    - For example we have the two tables (Employe with field id and Email with field id, employee id) where 1 employee has many emails and employee_id would be the value of joinColumn
      - Email entity will have a foreign key column named employee_id referring to the primary attribute id of our Employee entity.
    - with the @ManyToOne
    - Has all the information to create the relationship by defining the owningside
- mappedBy
  - @OneToMany(fetch = FetchType.LAZY, mappedBy = "employee")
  - private List<Email> emails;
    - the value of mappedBy is the name of the association-mapping attribute on the owning side

JPA Entity Relationships

- Example
- @Entity
- @Table(name="person")
- public class Person {
- @Id
- @GeneratedValue(Strategy.auto)
- private Long I
- @OneToMany(mappedBy=”person ”. cascade = CascadeType.ALL, fetch=FetchType.EAGER)
- private List<IdDocument> idDocuments;
- }
-
- @Entity
- @Table(name="ID_DOCUMENTS")
- public class IdDocument {
- @Id
- @GeneratedValue(Strategy.auto)
- private Long Id
- @ManyToOne
- @JoinColumn(person_id) @manyToOne
- private Person person;
- }
- Defines the entity that owns the relationship which is the Address entity in our case

JPA Entity Relationships

- Fetch Type
  - Lazy Fetch Type – Data is not queried until referenced
  - Eager Fetch Type – Data is queried immediately
  - Hibernate 5 supports JPA 2.1 fetch types
- Fetch Type Defaults for 2.1
  - OneToMany Lazy
  - ManyToOne Eager
  - ManyToMany Lazy
  - OneToOne EagerCascade Type
  - How state changes are cascaded from parent to child object
  - Types
    - Persist Save operations will cascade to related entities
    - Merge Related entities are merged when the owning entity is merged
    - Refresh Entities are refreshed when the owning entity is refreshed
    - Remove Removes all related entities when the owning entity is deleted
    - Detached All related entities if a manual detach occurs
      - Get an error if you try a lazy load
    - All Applies to all the above operations
  - By Default no operations is deleted.
- Embeddable Types
  - JPA/Hibernate support embeddable type
  - They are used to define a common set of properties (Example Address) and address is into a another class

JPA Entity Relationships

- Inheritance
  - MappedSuperclass – Entities inherit from a super class.
    - A database table is not created for the superclass
  - Types
    - Single Table (Hibernate Default) – One Table is used for all subclasses
    - Joined Table Base class and subclasses have their own tables. Fetching subclass entities require a join to the parent table
    - Table Per Class – Each subclass has its own table
- Create and update Timestamps
  - Best Practice to use create an update timestamps on your entities for audit purpose
  - JPA supports @PrePersist and @PreUpdate which can be used to support audit timestamps via JPA Licecycle callback
  - Hibernate provide @CreationTimestamp and @preUpdate which can be used to support audit timestamps via JPA lifecycle callbacks
    - Can use these annotation on the fields for creation / updated time stamp
  - Hibernate provides @CreationsTimestamp and @UpdateTimestamp

Recipe Data Model
Forking in Github

- JDL – From the Jhipster Team – https://start.jhipster.tech/jdl-studio/
- Forking in Github
  - The fork can be kept sync
    - Url : https://help.github.com/articles/fork-a-repo
    - The Subsection is “Keep your fork synced”
    - Summary : Add a remote, change to the project’s directory, git fetch upstream, git checkout master
  -

One to One Relationships

- Best Practice : Put your POJO in either a directory called domain or model
- @Entity → Indicates that it is a JPA entity and since no table exist the POJO is mapped to the same tablename
- Usually a Database String is 255 character use LOB for large characters
- Example
  - @Entity
  - Public class Recipe
    - @ID // Leakage : Since we are giving it a value
    - @GenratedValue(strategy=GeneratedType.IDENTITY) // Identity automatic id generation
    - Private Long id;
    - @OneToOne(CascadeType=CascadeType.all,mappedBy=”recipe”)
      - // If you delete a note you do not delete a recipe so, Notes should have joinColumn with th name of the
    - Private Notes notes; // for the Notes have a property recipe which has @OneToOne
    - setters/getters
    - @LOB // (L)arge (Ob)jects can be use for character or binate produce a CLOB field
    - Private String Description;
    - @LOG // Example for binary data using LOB
    - Private Byte image
- Note the Notes would have @OneToOne with a member Recipie
- With the Notes field we would have @OneToOne and @JoinColumn name (name of

One to One Relationships

- @OneToOne can have the fetch, cascade, mapped by, optional
- @JoinColumn used to specify the foreign key owner name, nullable

One To Many JPA

- Example
  - @Entity
  - Public class Recipe {
    - @Id
    - @GeneratedValue(strategy=GeneratedType.IDENTITY)
    - private long id;
    - private String description
    - private BigDecimal amount;
    -
    - // Sets up the relationship
    - // mappedBy – name of the filed that is the owning side
    - @OneToMany(cascade = CascaseType.ALL, mappedBy = “recipe”); // property on the ingredient class
    - Private Set<Ingredient> ingredients;
    - }
    - @entity
    - Public class Ingredient {
      - @ManyToOne
      - private Recipe recipe;
    - }

Many to Many

- Example
  - @Entity
  - Public class Category {
    - @ManyToMany
    - Private Set<Recipe> recipes;
    - }
    -
  - Public class Recipe {
    - @ManyToMany
    - Private Set<Categories> categories;
  - }
- If the @ManyToMany only is used for both the then two tables are created CATEGORY_RECIPE and RECIPE Category
  - Need to add @JoinTable(name = “recipe_category”, @JoinColumn(name=“recipe_id”), inverseJoinColumns = @JoinColumn(name = “category_id”)) to the categories property
    - From this class we are going to use recipe_id and then from categories we will be using category_id
  - In the Category Object
    - We Need @ManyToMany(mappedBy=”categories”) // Property Name from Recipe

JPA Enumerations

- Example
  - **Public enum Difficulty { EASY, MODERATE, HARD }**
  - In the Recipe class add private Difficulty difficulty and create the setters/getters
  - Example
    - @Enumerated(value=EnumType.String)
    - Private Difficulty difficulty
  - @Enumerated value name can have two values ORDINAL, STRING)
    - How it gets persisted in the database
    - ORDINAL is the default and will get persisted as 1,2,3
      - Problem as “kindOfHard” between easy and medium the numbering would be change where medium would be three instead of two
    - String get persisted as “easy”, “medium”, “hard:”

Creating Spring Data Repositories

- Best Practice Create a Repository Class
- Create a RecipeRepository, UnitOfMeasure, Category extends CrudRepository<Recipe, Long> // Recipe is an @Entity POJO
- The Classes should be the POJO_NAME + “Repository”
- The CrudRepository contains count, delete, exist, find, save functions
- The proxy can provide finder methods – Proxy will be implementations
  - naming is very importy <operation> (findBy) and name is a property of the table
  - Example – public interface EmployeeRepsoitory extends JpaRepository<Employee, Long>{ public List<Employee> findByFirstName(String name);
- org.springframework.data.jpa.Repository – Wrap the Entity Manager and provide a simple interface for JPA Based Access to relational databases
  - CrudRepository
  - PageAndSortingRepository
  - JPARepository
- At ApplicationContext Instantiation time Spring Boot recognizes these interfaces and generates a proxy implementation of your interface

Creating Spring Data Repositories

Keyword	Sample	JPQL snippet
And	findByLastnameAndFirstname	... where x.lastname = ?1 and x.firstname = ?2
Or	findByLastnameOrFirstname	... where x.lastname = ?1 or x.firstname = ?2
Between	findByStartDateBetween	... where x.startDate between 1? and ?2
LessThan	findByAgeLessThan	... where x.age < ?1
GreaterThan	findByAgeGreaterThan	... where x.age > ?1
IsNull	findByAgeIsNull	... where x.age is null
IsNotNull,NotNull	findByAge(Is)NotNull	... where x.age not null
Like	findByFirstnameLike	... where x.firstname like ?1
NotLike	findByFirstnameNotLike	... where x.firstname not like ?1
OrderBy	findByAgeOrderByLastnameDesc	... where x.age > ?1 order by x.lastname desc
Not	findByLastnameNot	... where x.lastname <> ?1

Creating Spring Data Repositories

- Inject the repository
  - public class EmployeeService implements EmplService {
    - @Inject
    - private EmployeeRepository employeeRepository;
    - public List<Employee> findByLastName(String lastName) {
      - String key = “%” + lastName + “%”;
      - return employeeRepository.findbyLastNameLike(key); }
- An extension of CrudRepository provides methods for paginatino and sorting
  - findAll(Pageable pageable) where the pageable object can contain properties used to specify page size, current page number and sorting
  - Example
    - public interface EmployeeRepository extends PagingAndSortingRepository<Employee, Long}
    - Example
      - PageRequest pageable = ReapgeRequest(0,5, sort.by(“dept”));
      - Iterable<Employee> sortedEmployees = repository.findAll(pageable);
      - sortedEmployees.forEach(System.out::println);
      - next page : repository.findAll(pageable.next()).forEach(System.out.println);

Creating Spring Data Repositories

- Which Repository should you extends
  - With CrudRepository the findById returns a optional
  - JPARepository returns T getOne(ID id)
    - Returns the entity without the Optional or throws an EntityNotFound
    - Control PersistenceContext like flush and context
      - execute SQL but not necessarily commit
- Using JPQL
  - provides named queries @NamedQery in domain classes , it does inject persistence specific information into a class
  - The named arguments require the @Param annotation on method arguments
  - Example
    - @Query(“Select em from Employee emp where dept = :department”) // Can define JPQL or native SQL Queries
    - Collection<Employee> getEmployees(@Param(“department”) long dept);
    - repository.getEmployees(300:L).stream.forEach(System.out::println);
- Bulk updates
  - must be in transaction
  - Example
    - @Modifying I dentify a select for update query
    - @Transactional
    - @Query(“Update Employee empl set empl.dept = :department where empl.id = :id”)
    - ‘int changeDepartment(@Param(“id”) long id, @Param(“department”) long department

Spring Data Rest

- REST Project provides a solid foundation to expose CRUD operations to your JPA Repository managed entities using plain HTTP Rest semantics
- Create REST representations of JPA entities that follow the Hypermedia as the engine of the application state of HATEOS principle
  - A rest client enters a rest application through a simple fixed URL or other exposed endpoints
  - All future actions the client may take are discovered within resources representation returned from the server
- The server returns a response that contains other URLs to other services
- Uses HAL Json with the content type application/hal+json for render response
- Spring Dependency – org.springframework.boot:spring-boot-starter-data-rest
- Minimal Implementation
  - Annotate the repository to expose it as a REST Server
  - Example
    - @RepositoryRestResource(collectionResourceRel=”staff”, path=”employees”)
    - pubic interface EmployeeRepository extends JpaReository<Employee, Long>
      - collectionResourceRel indicates the root element of the collection resource will be called in the seriali9zeed Json output from an endpoint path attribute
        - What the URL of the effective Rest Service will be

Spring Data Rest

- Hateos
  - Hypermedia as the engine of application state
    - We we perform a REST request we get the data and not any actions around it
  - Send the data and specifies the related actions
  - loose coupling
    - If you returns the URL then the actions can be changed as needed
  - embedded links to further actions or endpoints
-

Spring Data Rest

- To execute queries search is added to URL template to defined templated urls in the repository
- Question marks are place holders
-

Spring Data Rest

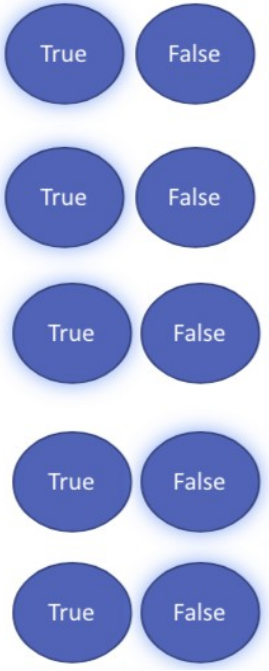
- When accessing your new Repository service in a browser you can click on these embedded actions or links urls to trigger the resultant service
-

Spring Data Rest

- Hal Explorer
  - A front end to browse your HAL Json.
  - Can point it at any Spring Data Rest and use it to navigate the app
  - Navigating to the root URL will redirect you the Hal Browser Home Page
  - Provide a convenient way to navigate through your service representations of your entities
- When modeling a bidirectional relationship we need a JPA Repository for our training entity so we can navigate to the Employee side and to Training side
  - Example
    - @RespositoryRestResource(CollecitonResourceRel-”trainingClassses”, path=”training”
    - public interface TrainingRepository extends JpaRespository<Training, Long> {}
    - @RespositoryRestResource(CollecitonResourceRel-”staff”, path=”employees”
    - public interface EmployeeRepository extends JpaRespository<,Employee, Long> {}
  -

Spring Data Rest

- The @RepositoryRestResource attribute collection*ResourceRel* indicates what the root element of a collection resource will be
- @Query applied to an abstract method ion a repository can declare a JPQL query
- A method in a Repository prefixed with “findBy” implies a SELECT in the underlying Criteria Query generation
- CrudRepository Provides overloaded methods for Pagination
- Spring Data Repositories do not recongnize JPA @OneToMany relationships



JPA Repositories – Projection

- Projections – returns a subset of the properties
- Create the projections class in the same package as the repository
  - Example Changing last name then have a class LastName
- Example
  - @Projection(name=”surname”, types = Employee.class)
  - public interface LastName { String getLastName(); }
- In order to use the projection we must use the following URL – http://localhost:8080/employees/9?project=surname
  - the get* dictates the JSON Element name
  - getLastName => lastName

- By default the Entity is not added to the projection, but that can be fixed and can use SPEL Express
  - @project(name=”surname” type=Employee.class)
  - public interface LastName {
    - String getLastName();
    - long getId();
    - @Value(“#{target.firstName.length()}”;
    - Long getFristNameLength(); }

JPA Repositories – Projection

- Excerpts
  - projections which apply as default views to resource collections
  - Adding the excerptProjection attribute to use our projection class
  - less data is represented and HATEOAS links led to details view of the entity
  - Example – Bring Back only the last name by bringing back projection.
    - @RepositoryRestResource(collectionResourceRel=”staff”, path=”employees”, excerptProject=LastName.class)
    - public interface EmploeeRepository extends PagingAndSortingRepository<Employee, Long> {}

Spring Data Rest

```
public interface EmployeeRepository extends JpaRepository<Employee, Long>{

}
```

a) Collection<Employee> findByLastNameAndAgeGreaterThan(String str1, int i1);
b) Collection<Employee> getLastNameAndAgeGreaterThan(String str1, int i1);
c) Query("SELECT x FROM Employee x where x.lastName = :y AND x.age > :z")
   Collection<Employee> getLastNameAndAgeGreaterThan(@Param("x")String str1, @Param("z)int i1);
d) Query("SELECT x FROM Employee x where x.getLastName = :y AND x.getAge > :z")
   Collection<Employee> getLastNameAndAgeGreaterThan(@Param("x")String str1, @Param("z)int i1);

a) Correct – findBy implies a SELECT and lastName age are properties of Employee
b) False – dynamic finders must start with findBy for a Collection select
c) Correct - the JPQL is defining the query from the entity Employee with names parameters
d) False - the properties getLastName and getAge are incorrect in the JPQL

Database Initialization with Spring

- Hibernate DDL Auto
  - DDL = DATA Definition Language Type of SQL for select (find tables and indexes, foregin key relationshsips)
  - DML = DATA Manipulation Language Type of SQL for insert, updates, delete
- Hibernate Property is set by the Spring Property spring.jpa.hiberante.ddl-auto
  - Allows the creation of tables from the @entity class
  - Options are none, validate, update, create, create-drop
    - Spring Boot will use create-drop fro embedded database (hsql, h2, derby) or none
    - Running in production use validate
    - Update use with caution if your JPA model changes or there is an error hibernate will use DDL statements to update the database
- Initialize with Hibernate
  - Data can be loaded from import.sql which is a Hibernate feature
  - Must be on root of class path and ddl-auto property is set to create or create-drop
-

Database Initialization with Spring

- Spring JDBC
  - Spring DataSource Initialize via Spring Boot will by default load schema.sql and data.sql from the root of the class
  - Spring Boot will also load from the schema-${platform}.sql and data-${platform}.sql
    - Must set spring.datasource.platform
  - May conflict with Hibernate's DDL Auto property
    - Should use setting of none or validate
    - Recommend use Spring Boot, but don't use Sprint DataSource Initializer
- Create a file call data.sql that contains insert statements in resource directory
- Hibernate has many naming stratergies such as CamelCase to DataName (All Uppercase and underscores)

Spring Data JPA Query Methods

- Creates a query based on the string contain in the method name
  - Advantage : Don’t write any SQL
  - Called Query Methods
- Example
  - public interface CategoryRepository extends CrudRepository<Category, Long> {
    - Optional<Category> findByDescription(String description);
  - }
  - @Controller
  - public IndexController {
    - private CategoryRepository categoryRepository;
    - public IndexController(CategoryRepository) {
      - this.categoryRepository = categoryRepository;
    - }
  - @RequestMapping({“”, “/”, “/index” })
  - Public String getIndexPage() { Optional<Category> categoryOptional = categoryRepository.findByDescription(“American”); }

Using Setters for JPA Bidirectional Relationships

- Example 1 recipe can have 1 note and 1 note can have 1 recipe
- Public setNotes(Notes notes) { // At the same time you set the notes set the relationship
  - this.notes = notes
  - notes.setRecipe(this); // Set Recipe
- }
- public Recipe addIngredient(Ingredient ingredient) { // Another Example of Abouve
  - ingredient.setRecipe(this)
  - this.ingredients.add(ingredient);
  - Return this;
  - }
- Common Design Pattern for JPA and keeps the code centralized. You do not have to worry about some one the initialization of the relationships

Spring Pet Clinic – Create Base Entity
Convert Owners to JPA Entities
Convert Vets to JPA Entities
Create Visitor Entity

- @MappedSuperClass – Designates a class whose mapping information is applied to the entities that inherit from it.
  - A mapped superclass no separate table defined for it
- .@GeneratedValue(strategy=GenerationType.IDENTITY)
  - Table Use an underlying database table to ensure uniqueness
  - SEQUENCE Use a Database Sequence (A Sequence Generator)
  - Identity Use a Database Identity Column and the database will provide the unique key, but tied to a specific database type
  - Auto picked by the persistence provider can give you some problems such as unpredictable results
- Convert Owners to JPA Specific Entities
  - @Column(name=”first_name”) – hibernate usually convert the properties to snake case ex. first_name) using the property name.
  - @Table(name=”Pets”);
- Convert Vets to JPA Entities
  - ManyToMany has a @Join that stores the ids of the table. Creates another table
-

Spring Pet Clinic – Create Visitor Entry
Spring Pet Clinic Add Spring Data JPA Repositories

- When using ORM prefer Objects over ID References
- For the Pet and Visit, we have Many Visits for one pet
  - @ManyToOne and @JoinColumn
- Spring Data JPA Repositories
  - Looking specifically at the CrudRepository
    - Other Repositories are PagingAndSorting and JPARepository
  - Created a directory repositories directly under root
  - Need to create a interface that extends CrudRepository<T, ID> where T is the Type and the ID is the
  - The CrudRepository will make these bean in the Application Context
- The Crud Repository has dynamic name for doing database calls

Add Spring Data JPA Owner Service

- Add Spring Data JPA Owner Service
  - The service layer abstract out the persistent store
  - We will run a Map or SQL Database
  - In the services create springdataapi directory
    - We need a class that implements OwnerService (A service for the owner of the pet) and has OwnerService as Interface
      - The methods are CRUD (findAll, save, delete)
    - For each method in the service add the calls to the database.
    - Interesting piece of code: ownerRepository.findAll(owners.:add))
    - Interesting Code
      - Optional<Owner> optionalOwner = ownerRepository.findById(aLong);
      - return optionalOwner.orElse(other:null);
    - @Profile(“springdatajpa”) // Makes the profile active
    -

Transactions

- Similar framework to JDBC, Hibernate, JPA
- Defined programmatically or declaratively
- PlatformTransactionManager implementation provides access to start/stop transactions
- Type of Transaction Manager such as JtaTransactionManager Maps to App Server Transaction
  - Hibernate Transaction Manager Maps to Hibernate Transactions
  - DataSourceTransactionManager Uses the RDBMS transactions
- Spring Delegates the underlying TransactionManager, it does not change the way any TransactionManager acts
- Transaction Manager
  - Source for transactions such as a DataSource and TransactionManager. Both get provided by SpringBoot when using a DataSoruce
  - In Application.properties javax.sql.DataSource and Spring Boot has registered an implementation of PlatfomTransacitonManager namely JpaTrasactionManager
    - Binds a Single JPA EntityManager to the thread of execution
  - Using a Transaction Manager abstracts the specifics of any particular TransactionAPI

```
spring.datasource.url=jdbc:derby://localhost:1527/StudentDB
spring.datasource.username=GUEST
spring.datasource.driver-class-name=org.apache.derby.jdbc.ClientDriver
```

Transactions

- Transaction scope
  - Comprised of all the beans in a Transaction
  - A transaction propagates from a bean to other beans that it invokes on and to resources used from a bean (methods)
  - Transaction Attributes and the current Transactional state determine the transaction context an invocation runs on
    - There are transaction attributes for propagation, isolation and timeout of transactions and for read ony transaction.
  - Starts a transaction or propagates an exiting transaction from one bean invocation to another under the control of our attributes
- Transaction Demarcation
  - In your code you can access a Transaction Manager that Spring manages. All the transaction managers implement the same methods
    - TransactionStatus getTransaction(TransactionDefinition definition) throws TransactionException
    - void commit (TransactionStatus status) throws TransactionException
    - void rollback (TransactionStatus status) throws TransactionException
  - Declarative allows Spring Configuration to control Transaction Boundaries
    - Smallest scope is usually method within a bean
    - Programmatic allows a finer level of control
    - A single statement can be wrapped in a transaction
    - This option has the least impact on your code and is most consistent with the ideals of a non invasive lightweight container
      - The AOP is hidden from the user. AOP uses proxying

Transaction

- Spring supports the following transaction attributes that control propagation of transactions
  - Mandatory Will use an existing TX, if present, otherwise throw an exception
  - Nested If a TX exists, marks a save point to rollback to other acts like required
  - Never Must execute outside any transaction, if a Tx is present it will throw an exception
  - NOT_SUPPORTED suspends an existing TX and start it again after execution completes
  - REQUIRED default will use an existing TX, otherwise it will create a TX
  - REQUIRES_NEW suspends an existing TX, if one exists always runs in its own TX
  - Supports Will use an exciting Tx others it run run without an TX
- Specified at both the bean and method level
- Propagation.REQUIRED is the default propagation
- Example
  - @Transactional(propagation = Propagation.REQUIRED) // Scope a Transaction to the method. Commits or rolls back at the end of the method
  - public void serviceMethod(long dept, long id) { // This triggers the container to proxy the bean that the method is in. Using Spring // transactional aspec much like around advice
    - Employee empl = repository.findById(id).get();
    - emp.setDept(dept); } // Commit

A simple update is shown below, retrieve the Employee, update its state in the cache of the EntityManager, synchronize with the database, flush the cache (generate SQL) and then commit the transaction whereby the database is updated with the current state of the Entity. Note the scope f the transaction is the start and end of the method, including any delegation inside the method to other classes

Transaction

- Declarative Rollback Rules
  - By default Spring will mark a TX for a rollback if an unchecked (runtime) exception is throw from the code executing a tx
    - Recommend way to trigger a rollback
    - The container will catch an unhanded exception as it bubbles up the call stack and will the TX for rollback
    - Checked exception will not , by default , result in a rollback
  - What exceptions will / will not cause TX rollbacks using the following @Transactional elements
    - Class noRollbackFor — Classes that should not trigger a rollback (RunTimeExceptions)
    - String[] noRollbackForClassName — Names of classes that should not trigger a rollback (RuntimeExceptions)
    - Class[] rollbackFor — Classes that should trigger a TX rollback (Checked Exceptions)
    - String[] rollbackForClassName — Names of classes that should trigger a RollBack (Checked Exceptions)
- Transactional Management Design
  - @Component
  - public class EmployeeService {
    - @Inject privateEmployeeRepository repository
    - @Trasactional (**propagation = Propagation.Required)** public void serviceMethod(Employee, long id, long dept, long otherId) {
      - repository.save(employee); repository.changeDepartment(otherId, dept); repository.blowUp(); } }

Transaction

- Our repository has been amended to have a default blowUp method.
  - Cause the running transaction to rollback.
  - Our update method will use any existing transaction if one exist or create new if invoked outside a transaction
  - Example
    - public interface EmployeeRepository extends PagingAndSortingRepository<Employee, Long> {
      - @Modifying @Transactional
      - @Query("Update employee empl set empl.dept = :department where empl.id = :id)
      - int changeDepartment(@Param("id") long id, @Param("department") long dept);
      - default void blowup() { throw new RuntimeException("bang!"); } // Runtime exception will rollback
    - }
- 

Transactions

- Example
  - public interface EmployeeRepository extends PagingAndSortingRepository<Employee, Long> {
    - @Modifying @Transactional(propagation = Propagation.REQUIRES_NEW)
    - @Query("Update employee empl set empl.dept = :department where empl.id = :id)
    - int changeDepartment(@Param("id") long id, @Param("department") long dept);
    - default void blowup() { throw new RuntimeException("bang!"); } // Runtime exception will rollback
  - }
  - The changeDepartment will happen in its own transaction and then resurrect the original transaction. Only transaction will be rollback
  - 

Transactions

- https://www.baeldung.com/transaction-configuration-with-jpa-and-spring
- Enabling Transactions
  - @EnableTransacitonManagement annotation
  - Spring Boot must include spring-data-* or spring-tx
  - Xml Configuration
    - <bean id="txManager" class="org.springframework.orm.jpa.JpaTransactionManager">
    - <property name="entityManagerFactory" ref="myEmf" />
    - </bean>
    - <tx:annotation-driven transaction-manager="txManager" />
- @Transactional support further configuration
  - propagation type , Isolation Level, Timeout, readOnly flag, Rollback rules
  - read only is very dangerous to use since it can be ignored by the transaciotn manager
- Transactions and Proxies
  - Spring creates proxies for all the classes annotated with @Transactional
    - Any self invocation calls will not start the transaction
    - Only public methods should be annotated with @Transactional – Any other visibilities will be silently ignored

Transactions

- Transaction Logging
  - org.springframework.transaction should be configured with a logging level of trace
- https://dzone.com/articles/how-does-spring-transactional
  - @Transacational
    - Consider the persistenceContext and databaseTransaction
    - Transactional Annotation – defines the scope of a single database transaction
    - persistentContext – A synchronizer that tracks the state of a limited set of Java object and makes sure that changes on those objects are eventfully persisted back into the db
  - When does an EntityManager span multiple databases
    - Using the Open

Transactions

We want the “delete” to always happened regardless of the outcome but the “save” should not be committed if the RuntimeException is thrown How would you amend the classes to achieve this? - Answer

```
@Transactional
public void add(@RequestBody Employee employee){
    employeeRepository.save(employee);
    employeeRepostiory.deleteById(1L);
    throw new RuntimeException("Blow Up");
}
```

```
public interface EmployeeRepository extends JpaRepository<Employee, Long>{

   void deleteById(Long id);

}  @Transactional(propagation = Propagation.REQUIRES_NEW)
```

Add MapSpace Visit Service
Spring Pet Clinic Using Spring Profiles for Configuration

- Creating the ServiceMap where we provide the implementation of all the “database” functions such as findAll, findById
- Creating an interface VisitorService extends CrudService<Visit,Long> that contains the Map or JPA
- Using Spring Profiles for Configuration
  - For the Map Instance of the Service and @Profile(“default”, “map”}) for each of the map classes
  - If no active profile is set then we fall back to the default profile up
  - Set Active Profile
    - In application.properties spring.profiles.active=springdatajpa

Product Lombok – features

- How Lombok works
  - Hooks in via the Annotation processor API
  - The AST (raw source code) is passed to Lombok for code generation before the java compile continues
  - Thus, produces property compiled java code in conjunction with the Java Compile
  - Unser the
  - target/classes' you can view the compile class files
  - Intellij wil decompile to show you the source code
- Project Lombok and IDE's
  - Since compile code is change and source files are not IDE's can get confused by this
  - More of an issue for IDEs serveral years old
  - Modern IDEs support Project Lombok for intellij verify you have enabled annotation processing under compiler settings
- Features
  - val local variables declared final
  - var mutable local variables
  - @NonNull Null Check will throw NPE if parameter is null

Project Lombok – Features

- @cleanup Will close() on resource in finally block
- @Getter Creates getter methods for all properties
- @Setter Create setter for all non-final proeprties
- @ToString
  - Generate String of classname and each filed separated by commas
  - Optional parameter to include field names
  - Optional parameter to include call to the super toString method
- @EqualsAndHashCode
  - Generates implements of equatls (Object Other) and hashCode()
  - By default will use all no-static non-transient properties
  - Can optionally exclude specific proerpties
- @NoArgsConstructor
  - Generates no args constructor
  - Will cause compiler error if there are final fields
  - Can optionally force which will initialize final fields with 0 / false /null

Project Lombok – Features

- @RequiredArgsContructor
  - Generates a constructor for all fields that are final or marked with @NonNull
  - Constructor will throw a NullPointerException if any @NonNull fields are null
- @All Args Constructor
  - Generates a constructor for all properties of class
  - Any @NonNull properties will have null checks
- @Data
  - Generates typical boilerplate for POJOs
  - Combines, @Getter, @Setter, @ToString, @EqualsAndHashCode, @RequiredArgsConstructor
  - No constructor is generated constructors have been declared
- @Value
  - The immutable variant of @Data
  - All fields are made private and final by default
- @NonNull
  - Set on parameter of method or constructor and a NullPointerExcepiton will be thrown if parameter is null

Project Lombok – Features

- @Builder
  - Implements the builder patter for object creation
  - ex. Person.builder(“Adam Savage”).city(“San Francisco”).job(“MythBusters”).job(“UnchainedReaction”).build()
- @SneakyThrows
  - Throw Checked exceptions without declaring in calling method’s throw clause
- @Syncronized
  - A safer implementation of Java’s synchronized
- @Getters(lazy = true) for expensive getters
  - Will calculate value first time and cache
  - Additional gets will read from cache
- @Log – Crate a Java util logger
- @Slf4j – Creates a generic logging facade
- Spring Boot’s default logger is LogBack

Adding Project Lombok
Using Project Lombok
Gotchas with Project Lombok

- Add the dependency to maven
  - Not putting in the version will get the curated version
  - The curated version come from the spring boot parent
- For Intellij load in the setting for project lombok to have the methods show up in the IDE
- Using Project Lombok
  - @Log → allows you to use logging statement right away.
- Gotchas with Project Lombok
  - Bidirectional Reference create circular references and produce a StackErrorOverflow
    - Fix use @EqualsAndHashCode(exclude = {“recipies”)) on Recipes and Ingredients an notes
  - Hibernate Lazy Initialization Exception
    - For the getRecipes() method – For getRecipes() we got out of th JPA Repositories, we are going out of a transaction and the layz collections need to get initialized in the transaction and the same session
      - Use @transacitonal around the method which will fix the above problem

Spring Pet Clinic – Refactoring for Project Lombok

- Lombok has some issues with inheritence
- The builder() allows you to build the whole at once
  - Needs an all args constructor and may need to create a chain of construcotrs
  - May need put the @Builder above the all args constructor
  - May not be able to use the all args constructor
-

Spring Pet Clinic – Amending Commit Messages

- git rebase -i Head~n
  - Where tilde is the previous commit and n is the previous commit
  - A message, in vi, will be displayed with instructinos
    - verify that the first hash is the same as the hash of the method that you want to commit
    - Change the first pick to reword
    - Change any other pick.
    - Save the file
- Next do a git push –force
-

Bootstrapping CSS

- Utilize the Bootstrap CDN
- For Intellij. If you have a tab with the browser icons an click on one then ti will show up in the browser
- Refresher
  - A div with class container file
  - A div for each row
  - Don’t forget to have a div with number of rows and the offset so the grid is centered.
    - There are 12 columns in a grid so if 6 are used then 3 will center it.
  - Panel-primary : provides a blue block around the header and the data.
  - classes
    - panel-title
    - panel-body
    - Table-hover
    - table-inverse
- Thyme leaf tag th:if=”$not #list.isEmpty(recipies)” which will not display the HTML Object where it resides and its children
- Thymeleaf
  - <tr th:remove=”all”><td th:text=”${recepie.id”}”>123<.tr></td><./tr> 0 recpies prints out dummy data will need one for each col
  - <tr th:each=”recipie : ${recpieis}”><tdth:text=”${recepie.id”}” >123</td></tr> >0 recpies prints out real data done only once

Testing the Spring Framework App

- Code Under Test – This is the code you are testing
- Test Fixture – A fixed state of a set of object used as a baseline for running test
  - Ensure there is a well know and fixed environment in which test are run to make the results repeatable
  - The text fixture is the test itself including input data, mock object, loading database with known data
- Unit Tests / Unit Testing – Code written to test code under test
  - Designed to test specific sections of code
  - Percentage of lines of code test is code coverage
    - Ideal coverage is in the 70 to 80 percent range
  - Should be unity and execute very fast
    - Targeting very specific code such as a class
  - Should have no external dependencies such as database or Spring context
- Integration Tests – Designed to test behaviors between object and parts of the overall system
  - Much larger scope
  - Can include Spring Context, database and message brokers
  - Will run much slower than unit test

Testing the Spring Framework App

- Functional Test – Testing a running application
  - Application is live, likely deployed in a known environment
  - Functional touch points are tested
    - Examples : web driver, calling web services, sending/receiving message
  - Examples of tools are Selenium and Spock
  - End to end Testing
- TDD – Test Driven Development
  - Write Test first which will fail then write the code to fix the test
- BDD – Behavior Driven Development – Builds on TDD and specifies that tests of any unit of software should be specified in terms of desired behavior of the unit
  - Often implemented with DSLs to create Natural Language Tests
  - Tools : Jbehave, Cucmber, Spock
    - Spock – Good framework for thinking about the test
    - Spock contains given (a set of data) , when (is the action of calling the method) , then (what do you expect to happen)
  - Open up the Test to a technically strong BA or QA person (DSL’s)

Testing the Spring Framework App

- Mociito – A java mocking framework
- JSONassert – An assertion library for JSON
- JSONPath – Xpath for JSON

- JUnit 4 Annotations
  - @Test Identifies a method as a test method
  - @Before Execute before each test used to prepare the test environment
  - @After Execute after each test. It is used to cleanup the test environment. Can save memory by cleaning up expensive memory structures
  - @BeforeClass Executed once before the start of all tests. Methods marked with this annotation need to be static
  - @AfterClass Executed once after all test have been finished. Methods annotated with this annotation need to be defined as static
  - @ignore Marks that the test should be disabled
  - @Test(expected = Exception.class) Fails if the method does not throw the named exception
  - @Test(timeout = 10) Fails if the method takes longer than 100 milliseconds
- Spring Boot Annotations
  - @RunWith(SpringRunner.class) Runs test with Spring Context
  - @TestConfiguration Specify a Spring Configuration for your test

Testing the Spring Framework App

- @TestConfiguration – Specify a Spring Boot Application for configuration
- @MockBean – Injects a Mockito Mock
- @SpyBean – Injects a Mockito Spy
- @JsonTest – Creates a Jackson or Gson object mapper via SpringBoot
- @WebMvcTest – Used to test web context without a full http server
- @DataJpaTest – Used to test data layer with embedded database
- @JdbcTest -- Like @DataJpaTest, but does not configure entity manger
- @DataMongoTest – Configures an embedded MongoDb for testing
- @RestClientTest – Creates a mock server for testing rest clients
- @Auto0configureRestDocs – Allows you to use SpringRestDocs in tests, creating API Documentation
- @BootStrapWith – Used to configure how the TestContext is bootstrapped
- @ContextConfiguration – Used to direct Spring how to configure the context for the test
- @ActiveProfles – Set which Spring Profiles are active for the test
- @TestPropertySource – Configure the property sources for the test
- @DirtiesContext – Resets the spring Context after the test (expensive to do)
    - Reset the Spring Context. When running Spring Context will get cached for multiple test. If you corrupted somehting it will rebuild the SpringContext
- @WebAppConfiguration – Indicates Spring should use a Web Application Context
- @TestExecutionListeners – Allows you to specify listeneras for testing event

Testing the Spring Framework App

- @Transactional – Runs test in transaction, rollback when complete by default
  - If you don't use the data will persisted anyway may corrupt another test
- @BeforeTransaction – Action to run before starting a transaction
- @AfterTransaciton – Action to run after a Transaction
- @commit – Specifies the transaction should be committed after the test
  - @Rollback – Transaction should be rolled back after test (default action)
- @Sql – Specify SQL scripts to run before
- @SqlConfig – Define meta data for SQL Scripts
- @SqlGroup – Group of @SQL annotations
- @Repeat – Repeat test x number of times
- @Timed – Similar to JUnit timeout, but will wait for test tom complete unlike Junit
- @ifProfileValue (rarely used) – Indicates test is enabled for a specific testing envorinment
- @ProfileValueSourceConfiguration (rarely use) – Specify a profile value source

Creating a JUnit Test

- For Spring Boot should have spring-boot-starter-test added as a dependency
- Example
  - @RunWith(SpringRunner.class) // Indicates that class should use Spring's Junit Facilities
  - // Joins the JUnit testing library with the Spring TextContext Framework
  - @SpringBootTest
  - Public class Spring5RecipieApplicationTests {
    - @Test
    - Public void contextLoads() {
    - }
  - }
- @ContextConfiguration – Class level metadata that is used to determine how to load and configure an ApplicationContext
  - declare the resource locations or the annotated classes that will be used to load the context
- Junit5 – Allows you to directly inject @inject, @Resource or @Autowried any dependencies into the Junit Class
  - @ExtendeWith(SpringExtension.class)
  - @ContextConfiguration(locations={classspath:beans.xml"}) or @ContextConfiguration(classes={AppConfig.class, AnoterhConfig.classs){

Using Mockito Mocks

- We are going to test a service routine which returns all the recipies
- Create a class with method annotated with @test
- Example
  - Public class RecipeServiceImplTest {
    - RecipeServiceImpl
    -
    - @Mock
    - RecipeRepository recipeRepository
    -
    - @Before
    - Public void setup() throws Exception {
      - MockitoAnnotations.intitMocks(this);
      -
      - RecipeService = new RecipieServiceImple(recipeRepository)
    - }
    -
    -

Using Mockito Mocks

- @Test
- public void getRecipes() throws exception {
  - // Setup the Recipe Data
  - Recipe recipe = new Recipe();
  - HashSet recipeData = new HashSet();
  - RecipeData.add(recipe);
  - // When the find all is called then return recipesData to the calling routine
  - when(recipeRepository.findAll()).thenReturn(recipesData);
  - Set<Recipe> recipes = recipeService.getRecipe();
  - assertEquals(recipe.size(), actual: 0);
  - verify(recipeRepository, times(wanted numberOfInnovations:)).findAll()
- }
- With Mock we want to verify actions
- Common Pattern : A service layer where you inject repositories and have business logic inside the service layer.

Mockito Argument Capture

- Standard behavior of Behavioral Driven Test : Given, When, Then
- Example
  - Public class IndexControllerTest {
    - @Mock
    - RecipeService
    - @Mock
    - Model model
    - @Before
    - Public void setUp() {
      - MockitoAnnotation.initMocks(testClass: this);
    - }

Mockito Argument Capture

- @Test
- public void getIndexPage() throws Exception {
  - // given
  - Set <Recipe> = new HashSet();
  - recipe.add();
  - recipe.add();
  - when(recipeService.getRecipes()).thenReturn(recipes);
  - Argument(Captor<Set<Recipe>> argumentCaptor = ArgumentCaptor.forClass(Set.class));
  - // Test the index
  - string viewName = controller.getIndexPage(model);
  - assertEquals(expected: “index”, viewName);
  - // test the recipe POJO
  - verify(recipeService, time(wantNumberOfInnovations, 1)).getRecipes();
  - verify(model, times(wantNumberOfInnovations: 1)).addAttribute(value: “recipes”), anySet());
  - Set<Recipie> setInController = argumentCaptor.getValue();
  - AssertEquals(expected: 2, setInController.size()); // Verify that we get a set back with two items
- }

Introduction to Spring Mock MVC

- Unit Test Spring Controller by provide a mock and bring in a mock servlet context
  - We are going throught a Mock Dispatcher Servlet
- Two Options Stand alone and Web App Context Setup
  - Web App Context Setup will bring up a context and that will cause our test to longer be a unit test
- Properties and Methods will go into the IndexControllerTest from the previous page
  - @Test
  - Public void testMockMVC() throws Exception {
    - MockMvc mockMvc = MockMvcBuilders.standaloneSetup(controller).build();
    - MockMvc.perform(get(urlTemplate: “/”)).andExcept(status().isOk()).andExpect(view()).name(expectedViewName: “index”));
  - }

Spring Integration Test

- In the Integration test use the context and the database
- Convention : IT for integration test
- @DirtiesContext → Causes the context to get reloaded (reset to default parameters)
- @DataJpaTest will bring up a JPA database and configure JPA
- @RunWith(SpringRunner.class)
- public class UnitOfMeasureRepository
  - @autowired
  - UnitOfMeasureRepository unitOfMeasureRepository
  - @Before
  - public void setUp() throws Excption {
  - }
  - @Test
  - public void findByDescription() throws Exception {
    - Optional<UnitOfMeasure> unitOptional = unitOfMeasureRepository.findByDescription(“Teaspoon”);
    - assetEquals(“expected: “Teaspoon”, uomOptional.get().getDescription())
  - }}

Maven Failsafe Plugin

- Maven will run anything with word test in it.
- Integration test are more expensive to run
- Maven will distinguish between test (it) and unit test (test)
- FailSafe Plugin will run the unit test which has a
  - group id of org.apache.maven.plugins and an artifcatId of maven-failsafe-plugin
  - For configuration
    - Include all test that end in IT.java
    - Need to add target/class
    - Can set the Parallel option
  - For execution
    - Goals are integration-test and verify

Continuous Integration Testing with Circle CI

- A free service for integration builds. A docker container that will integrate with github
- Every time a commit happen CircleCI will do a build
- Website is circleci and login and authorize circleci to your github account
- Use follow and Build to run a build
  - In a container, so it needs to get the artifacts from maven
- Advantage is circle knows about branches and will do testing with those.
- CircleCI configures a github webhook to know when the user has commited

Junit 5

- Junit vintage which allow the execution of Junit3 and Junit5
  - Easy migration to Junit 5
- Difference
  - Instead of @Test(expected=Foo.class) use Assertions.assertThrows(Foo.class)
  - Instead of @Test(timeout=1) use Assertions.assertTimeout(Duration = 1)
  - Instead of @RunWith(SpringJUnit4ClassRunner.class) use @ExtendWith(SpringExtension.class)
  - Different annotations (Junit4/Junit5)
    - @Before/@BeforeEach @After/@AfterEach @BeforeClass/@BeforeAll @AfterClass/@AfterAll
    - @Ignored/@Disabled @Category/@Tag
- Other testing tools : Spock , Test Containers,

Spring Pet Clinic – Convert To Junit 5

- Convert from Junit4 to Junit5 for SpringBoot (May not be necessary as of 02/05/2020
  - Modify the POM Files
    - For the maven coordinates: org.springframework.boot, spring-boot.start-test
      - Add an <exclusion> for maven coordinates:junit, junit
    - Add two dependencies: junit-jupiter-api, junit-jupiter-engine and scope of test dependency
    - Setup the Maven sure fireplugin
      - Add plugin, artifact id → maven-surefire-plugin and version
        - Add dependency org.junit.platform, junit-platform-surefire-provider with the junit-platform-version as the ${junit-platform.verion}
          - The junit platform.version will be the version of surefire from the SpringBoot Parent Pom
- Example
  - @ExtendsWith(SpringExtension.class)
  - @SpringBootTest
  - public class SpringPetClinicApplicationTest {
    - @Test
    - Public void contextLoads() {}
  - }

Spring Pet Clinic – I18N French Message Properties

- Looks at the request header and search the accepted language
  - The accept language en-US; en;q=0.9
  - Alerts to use message_en.properties. If you wanted specific message_en-US.properties
- Possible to setup a locale change interceptor, but Spring Boot does not configure it by default

Spring Pet Clinic – CRUD Tests for Owner Map Service

- Testing the OwnerMapService
- Example
  - Class OwnerMApServiceTest {
    - OwnerMapService ownerMapService
    - // Creates and initializes an owner map service
    - void setUp() {
      - OwnerMapService = new OwnermapService(new PetTypeMapService(), new PetMapService());
      - OwnerMapService.save(Owner.builder().id(1L).build();
    - }
    - @Test
    - void findAll() {
      - set<Owner> ownerSet = ownerMapService.findAll();
      - assertEquals(expcected: 1, ownerSet.size());
    - }
  - }
  - Interesting Stream : this.findAll().stream().filter(owner → owner.getLastName().equalsIgnoreCase(lastName)).findFirst().orElse(owner: null);
    - this.findAll() → gets Set<Owner>

CRUD Test for Owner SD JPA Service

- Mockito-Junit-Jupiter is the maven artifact need for mockito/junit integration
  - This is not in the parent pom for Spring Boot so will need to include parent information
- For the owner SDJPA Service mock out the OwnerRepository, PetRepository, PetTypeRepository
  - @Mock – Create a mock object that can be injected
- For the OwnerSDJPAServeice @injectMocks
  - Create and inject a mock object
- Example Test
  - Owner returnOwner = Owner.builder.id(1l).lastName(“Smith”).build()
  - when(ownerRepository.findByLastName(any())).thenReturn(returnOwner)
  - Owner smith = serviceFindByLastName(“Smith”)
  - assertEquals(LAST_NAME, smith.getLastName());
  - verify(ownerRepository).findByLastName(any()); // Assures whether the mock method is being called
-

Spring Pet Clinic – Testing Owner Controller With MockMVC

- Use the BeforeEach to
  - create a default set of Owners. It will be recreated after each test is completed
  - mockMvc = MockMvcBuilders.standaloneSetup(controller).build
- @Test
  - Void listOwners() {
    - when(ownerService.findAll().thenReturn(owners);
    - MockMvc.perform(“/owners”).andExpect().isok())
      - .andExpect().name(“ownersindex”)
      - .andExpect(model().attribute(“owner”)
      - .andExpect.attribute(hasSize(2)
  - }
  - @Test
  - Void findOwners() throws Exception {
    - mockMvc.perform(get(“/owners/find”))
      - .andExpect(status().isOk())
      - .andExpect(view().name(“notimplemented”);
    - verifyZeroInteractions(ownerService);
  - }

Using Webjars with Spring Boot

- WebJars are client side web libraries (Jquery and Bootstrap) packaged into JAR files
  - Jar File available in maven repositories of popular web components
  - In the POM File add the BootJar and Jquery and add them a dependency
  - In the Thymleaf add the link which should contain the following link for the bootstrap min css and the script for boostrap min js
    - See the code for an example
  -

Spring Pet Clinic – CI with Open CI

- Add Project sfg pet clinic
  - Go into Add Project → maven
  - Copy to clipboard and edit locally
- In the file
  - We have a build: job
    - Our build will run inside a container with JDK 8
    - Working directory is ~/repo
    - Steps currently Checkout
    - Cache dependencies and can save the cache
    - Can run tests instead of mvn integration-test
      - The artifact was not installed spring pet clinic data
      - Add –run: mvn install -DSkipTests which tells maven to download everything (this will be -run is called which calls the test)
        - Installs the modules
  - Commit to directory
  - Click the build button to test the build
  - For GitHub you can #63 on commit so you can trace them
  - Circle update the Java Image use JDK 11

Circle CI Build Badge
Spring Pet Clinic – Bug JUnit5 test not running from maven

- Goto CircleCI and select the project then status badges
- Select the Status Badge – Markdown was selected
- Edit the readme file and make sure you have the build where you want it and commit
- Can add #64 to trace it without closing the enhancement
- You will see a green checkmarks for the commits that passed
- Spring Pet Clinic – Bug Junit5 test not running from maven
  - Circle CI maven was not running the test
  - Need to remove dependency on the surefire plugin for the JUnit Version
  - Still need to add the maven-surefire-plugin artifact unless you have a version of maven > 3.5.4
    - SpringBoot provide you with a mvn wrapper verify that version
    - Another concern circle ci has its own version of maven
  - Intellij will be using the maven version found in the wrapper

Display a Recipe ID

- Another Tag : th:href="@{'/recipe/show' + ${recipe.id}) // Passing in a relative URL
- Bootstrap
- In the RecipeService we are adding
  - FindById which will find the recipe by id and if it is not present then it will throw a runtime exception
- In the RecipeController
  - Add a function showByID
  - @RequestMapping("/recipe/show/{id}")
  - Public String showById(@PathVariables String id, Model model) {
    - model.addAttirbute("recipe")
    - return "recipe/show"
    - }

Processing Form Post with Spring MVC

- Posting data back to the data for the initial creation and update.
- Lets say we have a web controller and controller
  - In the controller we have an object that we bind data to
    - Could be used in a update or a creation where you want to send back object with data
  - The object is padded to the web application
    - The object get submitted back by post
    - The post is form-urlencoded which is &,=
  - The form property name must be the same as the object property name so the properties
- DataBinding in Spring
  - Command Objects (aka Backing Beans) are used to transfer data to and from web forms
  - Spring will automatically bind data or form posts
  - Binding done by property name (less get/set)
-

Processing Form Post with Spring MVC

- Example of a PersonBean
  - firstName would bind to property firstName
  - Address.addressLine1 would bind to the addressLine1one of the address property
  - email[0]/email[1] would bind to index zero and one of the email List or Set property of Person
-

Creating a command Object and Type Conversion in Spring

- Need a directory called commands
- Example
  - @Setter
  - @Getter
  - @NoArgsConstructor
  - Public class RecipeCommand {
    - private Long id;
    - private String description
    - Private list<Ingredients>
  - }
- Wisdom: Smaller project expose the domain object and use those to bind to web forms and then evolve to use command objects
  - As the project evolve there may be some information in the domain object that you don't want the front end to access maybe ssn
  - As your project grows, the requirement of your domain objects are going to be different the requirements for the web tier
    - Example Credit Card Number : The DTO can have the full created card number and the
- Spring does not guarantee thread safety on converters so synchronize the method

Creating a command Object and Type Conversion in Spring

- Need an other directory @Converters
  - @Component
  - Public class UnitOfMeasureCommandToUnitOfMeasure implements converter<UnitOfMeasureCommand, UnitOfMeasure> {
    - @Synchronized
    - @Nullable
    - @Override
    - public UnitOfMeasure convert(UnitOfMeasureCommand source) {
      - If (source == null) { return null }
      - final UnitOfMeasure uom = new UnitOfMeasure();
      - uom.setId(source.getId());
      - uom.setDescription(source.getDescription());
      - return uom; }}
  - In Unit Test we will need to wire the converters
- In the RecipeServiceImpl we call the convert function save(RecipeCommand) where the first line will call the convert the Web Object to the Domain Object
  - Think of it as a detachedObject
  - Now made the save routine transactional → The Converts are outside the Spring Context
- The Command Object is passed into the controller then the service where it is converted from a command to a DTO

Creating a command Object and Type Conversion in Spring

- For integration Test the @SpringBootTest bring up the whole SpringCotnext. The @DataJpaTest brings up a lighter test
- Test Drive Development
  - Given Get the data and simulates what the front end would do
  - When Run the command
  - Then Verify the results are correct.
-

Overview of Exception Handling

- HTTP 500 – Internal Sever Error – Generally any unhandled exception
- HTTP 400 – Client Errors generally checked errors
  - 400 Bad Request – Cannot process due to client error
  - 401 Unauthorized – Authentication required
  - 404 – Resource not found
  - 405 – Method Not allowed
  - 409 – Conflict (Possible with simultaneous updates)
  - 417 – Expectation Failed – Sometimes used with Restful interfaces
    - A property missing from JSON
  - 418 I am a tea pot – April Fools joke from IETF in 1998
- @Response Status – Allows you to annotate custom exception classes to indicate to the framework the HTTP status you want returned when the exception is thrown
  - Global to the application

Overview of Exception Handling

- @ExceptionHandler works at the controller level for a specific exception type
- Allows you to define custom exception handling
  - Can be use with @ResponseStatus for just returning a HTTP Status
  - Can be use for a specific view
  - Also can take total control and work with the Model and View
    - Model cannot be a parameter of an ExceptionHandler method
- Way to handle an error
  - Might want to create an error page
  - Might want to hide the error details and make that visible when there is an error
  - Might want to make a modal for a lot of details
  - In production a clean error string should be compared.
- HandlerExceptionResolver
  - An interface you can implement for custom exception handling
  - Used internally by Spring MVC, the model is not passed
  -

Overview of Exception Handling

- Public interface HandlerExceptionResolver {
  - @Nullable
  - ModelAndView resolveException(HttpServletRequest request,
    - HttpServletResponse,
    - @Nullable Object Handler
    - Exception ex);
  - WorkingModel is not passed into. So we can return back an model and view.
    - Has some limits
- Spring MVC has 3 implementations of HandlerExceptionResolver
  - ExceptionHandlerExceptionResolver – Matches uncaught exception to @ExceptionHandler
  - ResponseStatusExceptionResolver – Looks for uncaught exceptions matching @ResponseStatus
  - DefaultHandlerExceptionResolver – Converts standard Spring Exception to HTTP status code (internal to Spring MVC

- You can provide your own implementations of HandlerException Resolver (Usually don’t need to go to this level)
  - Implemented with Spring’s ordere interface to define order the handlers will run
  - Custom Implementations are uncommon due to spring robust exception handling

Overview of Exception Handling

- SimpleMappingExceptionResolver – Handle for thrown exception where you want to direct the user to a specific page
  - A spring bean you can define to map exceptions to specific views
  - Define the exception class name (no package) and the view name
  - You can optionally define a default error page which is static and you cannot pass data to.
- Which to use when
  - If just setting the HTTP status use @ResponseStatus
  - If redirection to a view use SimpleExceptionResolver
  - If setting the HTTP status and @ExceptionHandler on the controller

Using Spring MVC Annotation @ResponseStatus

- Example
  - @ResponseStatus(HTTPStatus.NOT_FOUND) // THE HTTP Status when the exception is thrown. Default is 500
  - Pubic class NotFoundException extends Runtime Exception {
    - public NotFoundException() {}
    - public NotFoundException(String message) { super(message); }
    - public NotFoundException(String message, Throwable cause) { super(message, cause); }
  - }
- Have a JUnit test verify that a NotFoundException is being thrown
-

Showing Error Data on 404 Error Page

- In the recipe controller
  - Example
    - @ResponseStatus(HttpStatus.NOT_FOUND) // Without this the actual HTTP Code is 200
    - @ExceptionHandler(“NotFoundException.class”) // For our custom exception
    - Public ModelAndView handleNotFound() {
      - log.error(“Handling not found exception”);
      - modelAndView modelAndView = new ModelAndView();
      - modelAndView.setViewName(“404error”); // Matches the 404 html page
      - return modelAndView
    - }

Spring MVC Controller Advice

- Setup a Global Type Exception at the global level instead of each controller
- Example – Replaces the code in the controller
  - @Slf4j
  - @ControllerAdvice
  - Public class ControllerExceptionHandler {
    - @ResponseStatus(HttpStatus.NOT_FOUND) // Without this the actual HTTP Code is 200
    - @ExceptionHandler(“NotFoundException.class”) // For our custom exception
    - Public ModelAndView handleNumberFormat() {
      - log.error(“Handling number format exception”);
      - log.error(exception.getMessage());
    -
      - modelAndView modelAndView = new ModelAndView();
      - modelAndView.setViewName(“400error”);
      - ModelAndView.addObject(“exception”, exception)
      - return modelAndView
    - }
  - In the Unit we are using less then the full integration need use add mockMvc.setControllerAdvice(new ControllerExcepitonHandler);
  -

Data Validation with JSR 303

- Primary focus to define annotations for data validation
  - First release dealt with field level properties
  - Second release method level, validation to validate input parameters and Includes dependency injection for bean validation components
  - The third release was Java 8 language features and add 11 new build in validation annotation
  - Hibernate Validator 6.0 may be the only validation api.
- Built in Constraint Definitions
  - @Null, @NotNull, @AssertTrue, @AssertFalse, @Min, @Max
  - @DecimalMin, @DecimalMax, @Negative, @NegativeOrZero, @Postive, @PositiveOrZero,
  - @Size (check if string or collection) is between min and max
  - @Digits – Check for integer digits and fraction digits
  - @Past, @PastOrPresent,@Future, @FutureOrPresent
  - @Pattern
  - @NotEmpty – Checks if value is null or empty
  - @NonBlank – Checks String is not null or whitespace characters
  - @Email

Data Validation with JSR 303

- Hibernate Specific Validation not part of the Bean Validation as the previous
  - @ScirptAssert class level annotation checks class against script
  - @CreditCardNumber
  - @Currency
  - @DurationMax – Duration less than given value
  - @DurationMin – Duration greater than given value
  - @EAN –Valid EAN Barcode
  - @ISBN valid ISBN value
  - @Length – String length between a min and max
  - @CodePointLength – Validate that code point length of the annotated character sequence is between min and max included
  - @Luhncheck – Luhn check sum
  - @Mod10Check – Mod 10 Check sum
  - @Mod11Check – Mod 11 Check sum
  - @Range – checks if number is between given min and max inclusive
  - @SafeHtml
  - @UniqueElements
  - @Url
- Can define your own validation

Data Validation with Spring MVC

- Example
  - @Getter
  - @Setter
  - @NoArgsContstructor
  - Public class RecipeCommand() {
    - @NotBlank
    - @Size(min=3, max=255)
    - Private String Description
  - }
- By using these validators the application will not launch if the value are invalid and will get error messages printed out

What is Docker

- Docker can easily bring up components such as Mysql, Postgres
- A standard for linux containers
  - A container is an isolated runtime inside of linux which provides a private machine like space under linux
  - Have their own process space
  - Have their own network interface and their own disk space (can be shareable with the host)
  - Run processes as root
  - Using the host operating system
- A virtual machine is written in software that has a guest operating system which takes up a lot of resources
- Each container has its own libraries and applications and a they are firewall off.
- Speed comparison is much more efficient Docker could be 30 to 40 faster
- Docker Image – The representation of a docker container (like a Jar or War file).
- Docker Container – The standard runtime of Docker. Effectively a deployed and running Docker image
- Docker Engine – The code which manages Docker stuff. Creates and run Docker Images.

What is Docker

- An engine that enables any payload to be encapsulated as a lightweight, portable, self-sufficient container
- Standard operations and run consistently on virtually any hardware platform.
- Docker Engine Run Time (From lowest to highest)
  - Server (Docker Daemon)
  - Rest API
  - Docker Cli – manages network, data volumes and container and image
    - Can talk to other machines running Docker
-

Docker Editions

- Docker Enterprise Edition
  - C(ontainer) A(s) (A) (S)ervice platform subscription
  - Enterprise class support
  - Quarterly Releases and Backported packages for up to one year
  - Certified Infrastructures such as containers
- Docker Community Edition
  - Monthly ‘edge’ releases with latest features for developers
  - Quarterly release for operations
- Docker Release : number of year dot month dot version dot edition yy.mm.v.ed (Example v17.10.)
-

Hello World With Docker
Docker Hub
KiteMatic

- docker run hello-world
  - If it does not find it locally then it will pull down from docker hub if it can find it
  - Show a message to prove docker is working
- Docker will cache them locally
- Docker Hub
  - A public docker register where you can download images
  - hub.docker.com
  - In the docker if you leave the tag empty you get the latest and if you supply a version that version is use
- KitMatic
  - A GUI Tool for running images for docker hub
  - Settings: container id, environment variables, show the local port and remote port

Running Mongo DB Docker Container

- docker pull mongo
- docker run –name some-mongo -d mongo
- docker ps
- docker run -d mongo
- docker stop
- docker run -p 27017:27017 -d mongo // Changes the port number to 27017
-

Docker Images

- An image defines a Docker Container
  - Similar concept to a snapshot of a VM
  - Or a class vs an instance of the class
- Image are immutable
  - Once built, the files making up an image do not change
- Images are built in layers
  - Each layer is an immutable file, but is a collection of files and directories
  - Layers receive an ID, calculated via a SHA 256 hash of the layer contents
    - If the layer changes then the SHA 256 changes
- Docker image input mongo // See the layers that are make up the mongo image
- Image IDs
  - A SHA 256 hash derived from the layers. If the layers change the hash value changes
  - Th image id list by docker commands is the first 12 characters of the hash
- Docker images -q –no-trunc

Docker Images

- The hash values of images are referred to by tag names
  - ex. ip address vs domain name
- docker images // can see the tag name
- Image Tag Names
  - The format of the full tag name is [REGISTRYHOST/][USERNAME/]NAME[:TAG]
  - For register hub.docker.com is inferred
  - For :TAG latest is default
  - Full tag example: register.hub.docker.com/mongo:latest

Docker Files

- A recipe for the image
- from debian:whisky // Runs (Inherits) form this file
- Use the run command to execute commands
  - run executes commands
  - Each run will create a new image layer
  - Example
    - RUN app-get update \
      - && apt-get install -y –no-install-recommends \
      - numactl \
      - && rm -rf */var/lib/apt/lists/**
- *Different Docker Images that are dependent (inherit files) (think from debian:whisky) will use those layers form the dependent files that are same and not create their own.*
- *The Docker Daemon will make a container out of the image.*
- *When running container you will be running different containers.*

Non Persistent Container Storage

- The very last layer is allowed to be read/write while the previous layers are read only
- docker stop <Hash Code>
- docker run -p27017:27017 -d mongo
- docker ps
- docker logs -f <hash code>
- The persistent data does not make it across a persistent restart. The last layer got create again when the container started up
  - This layer is being written to these chunks can start accumulating space there are some commands used to clean up the images

Assigning Storage

- Assign storage on the host for the contain to use
- Use docker with the -v option to map the host directory to the container direct
- docker run -p 27017:27017 d mongo
- mkdir dockerdata/mongo // The place where the container storage will be stored
- docker run -p 27017:27017 -v /*Users*/jt/dockerdata/mongo:/data/db -d mongo // The -d mongo always has to be the last item
-

Docker House Cleaning

- With Development use Docker can leave behind a lot of files which will grow and consume a lot of disk space
  - Less of an issue on production systems where containers are not being build and restarted all the time
- Three key area of house cleaning: Containers, Images, Volumes
- Containers
  - Docker will not delete the containers until the container is stopped.
  - Kill all running Docker Containers: docker kill $(docker ps -q)
  - Delete all stopped docker containers: docker rm $(docker ps -a -q)
- Images
  - Built another image then it will be untagged (without a tag)
  - Remove a Docker Image : docker rmi <image name>
  - Delete Untagged (dangling images) docker rmi $(docker images -q -f dnagling=true)
  - Delete All Images: docker rmi $(docker iamges -q)
- Volumes
  - Once a volume is no longer assoicated with a container it is considered dangling.
  - Remove all dangling volumes: docker volume rm $(docker volume ls -f dangling=true -g) // Does not remove files from host sytstem in shared volumes

Preparing CentOS for Java Development
Run your own spring Boot App
Sample Spring Boot Application

- docker run -d centos
- docker ps
  - May not show any information. The Centos Image does not have a command. Docker containers run until the last command is completed. If the has no command then it will end
    - Fix: docker run -d centos tail -f /*dev*/null
- docker -exec -it naughty_brattain bash // Get a shell into the docker container
- Run Your Own Spring Boot App :
  - Spring boot give you an fat (all the jar files are included) executable jar
- Sample SpringBoot Application
  - Interface Application<ContextRefreshedEvent> – An hook into an event where the data is loaded. Use the OnApplicationEvent(ContextRefreshedEvent) is part of the interface
  - Spring has a HttpSecurity that can authenticate any request
  - SpringBootWeb provides the Tomcat

Running Spring Boot Application from Docker

- Example Docker File
  - From centos
  - RUN yum install -y java
  - Volume /tmp — Spring Boot will store tmp files
  - Add /spring-boot-web-0.0.1-SNAPSHOT.jar myapp.jar — Add from the current directory as myapp.jar
  - RUN sh -c ‘touch /myapp.jar’ — Updates the date on the jar import for static resources
  - EntryPoint [“java”, “-Djava.security.egd=file/dev/./urandom”, “-jar”, “/myapp.jar”]
    - EntryPoint – A command docker image will run when it is run inside a container
    - Helps Tomcat to startup faster
- docker build -t spring-boot-docker .
- Each one of these steps will produce a layer
- docker run -d p 8080:8080 spring-boot-docker

Introduction to Mongo Db

- Document orientated database
- Hu(mongo)us → extremely large
- Developed in C++
- MongoDB is a NoSQL Database
- MongoDB documents are stored in BSON
  - Binary JSON
- Why would you use MongoDB
  - Great for high insert systems
    - Examples : Sensor Readings, Social Media, Advertising Systems
  - Good when you need schema flexibility
    - Can add properties dynamically the result is easier to change
  - Can also support a high number of read per second
- cluster mongo servers, distribute their load and create replica servers

Introduction to Mongo Db

- Pitfalls
  - MongoDB has no concept of transactions
    - No ACID (Makes it faster)
    - no stomping on each other from other threads or sessions
  - No locking for transaction support hence faster inserts
  - Not good for concurrent updates
  - Could lose data ???
- Mongo DB Terminology (RDMS/MongoDB)

RDMS	MongoDB
Database	Database
Table	Collection
Row	Document
Column	Field
Table Join	Embedded Documents
Primary Key	Primary Key
Aggregation	Aggregations Pipe Line

- https://dzone.com/articles/why-mongodb

Mongo Application Code Review

- Purpose : Port the recipe applcation to Mongo
  - 1st Change for spring boot : com.springframework.bootspring-boot-start-data-mongodb
    - swapped out JPA for Mongo
  - In Relations the Id would be a long, but in mongo we made it a string
  - Cannot use the JPA Annotations for validation
  - For testing : Handle the change of the ID to a String
  - The Controller Layer stays the same, The Controller Layer Stays the Same, The Repository Layer Stays the Same

Circle Configuration
Code Coverage

- New Circle CI 2.0
  - image: circle/openjdk:8-jdk
    - should have maven and gradle
  - between build restore_cache and save_cache
- Code Coverage
  - apply the plugin : jacoco
  - add the configuration for jacoco to maven/gradle. Exaple which reports html, xml
  - Need to run curl -s https://codecov.io/bash to send the result to the website.
  - For code coverage : 75 to 90 %
  - Can also get a badge.

Embedded Mongo Database

- To add the database
  - de.flapddodle.embed:de.flapddole.embed.mongo // Compile dependency for the spring component
  - group czlirutka.spring, name embed-mongo // spring component for mongo
- In application.properties
  - spring.data.mongo.port
  - spring.data.mongo.host
-

Refactoring Data Model for MongoDB

- Example – Convert a POJO for mongo db use
  - @Get
  - @Setter
  - @Document
  - public class UnitOfMeasurement {
    - @Id
    - private String id;
    - private String description
- @Document – This annotation marks a class as being a domain object that we want to persist to the database:
- embedded mongo is running on its own port
  - RoboMongo – Edit the port and you can see the
- With Mongo you are not using Bidirectional ID
  - Under Mongo it is more national object model
  - JPA : The Persistence Layer is influencing you object design.

Correcting Application Defects under Mongo DB

- Update a value and the save it. The Categories and Ingredients
  - Under JPA we were doing a merge and not having the complete data and mongo does not do the merge the way that
  - Under Mongo : Pass in the Ingredients into a hidden list so the ingredients are there in the thyme leaf
  - Refactor from a set to an arraylist: Spring MVC does not bind to a set, but to a list
- The Ingredient is a nested list property
  - Fix created an id value
  - Mongo is a document database and does not need the array so we added a unique to

Integrating Testing with Mongo DB

- Integration Testing with MongoDB
  - Replace the @JPADataTest with @DataMongoTest with
  - The test is failing since we are not getting any values out of the database
  - To the test need to add the category and recipe and UnitOfMeasureRepistory and then @autowire then we call the bootstrap function
- Returning a non unique result
  - Run a deleteAll on the following to reset the database
    - UnitOfMeasureRepository
    - Category
    - Recipe
  - In JPA we have transactions and Spring rolls back after every test
  - Could have set a special test configuration, but it may not be better
- Leakage : Going from the backend to the front end
-

Reactive Manifesto

- www.reactivemanifesto.org
- Functional Reactive Programming
- Reactive Manifesto looks at 4 areas
  - Responsive
    - System response in a timely manner which is the corner stone of usability and utility
    - Problem may be detected quickly and dealt effectively
    - Provide rapid and consistent response times
      - Simplifies error handling, builds end user confidence and encourages further interaction
  - Elastic
    - They system stays responsive under varying workload
    - Can react to changes in the input rate by increasing or decreasing resources allocated to service inputs
    - Achieved on commodity hardware and software platforms
      - Dynamic scaling servers start up as needed

Reactive Manifesto

- Message Driven
  - Rely on asynchronous message passing to establish a boundary between components
    - This ensures loose coupling, isolatino and location transparency
  - Message passing enables load management, elasticity and flow control
    - Can go over rabbitMQ, JMS, Web Service and passing data JSON, XML or more across some boundry to another system
  - Location transparent messaging makes management of failures
  - Non-blocking communication allows recipients to only consume resources while active leading to less system overhead
    -
- Resilient
  - Systems stays responsive in the face of failure
  - Resilience is achieved by replication, containment, isolation and delegation
    - Failures are contained within each component
    - Isolation – Parts of the system can fail without compromising the system as a whole
    - Recovery of each component is delegated too another
    - High Availability is ensured by replication where necessary
- Using Spring Data with Mongo is fairly transparent (usually capability with dynamically adding new variables)

Reactive Manifesto

- Reactive Programming with Reactive Systems
  - Focuses on non-blocking asynchronous execution
  - Just one tool in building Reactive System

What is reactive programming

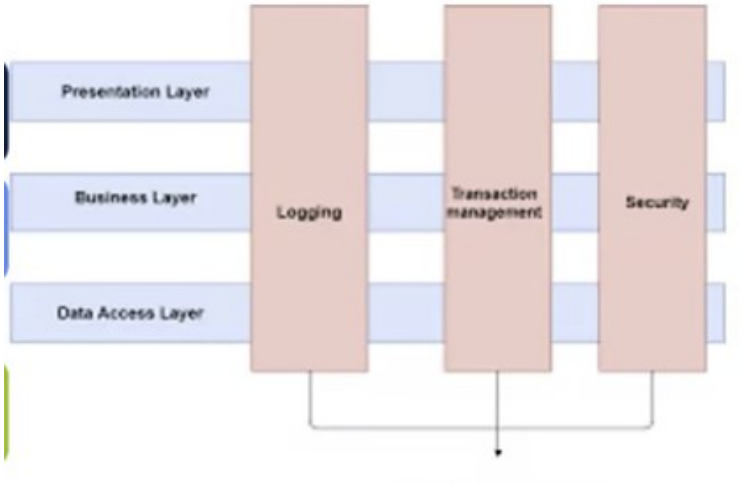
- Reactive programming is asynchronous programming focues on streams of data
- Reactive programs
  - Maintain a continuous interfaction with their environment, but at a speed which is determined by the environment not the program itself
  - Programs work at their own pace and mostly deal with communication which reative program only work in response to external demands and mostly deal with accurate interrupt handling
  - Real-time programs are usually reactive
- Processing a stream of events
- Common use case
  - External service calls
  - Highly concurrent Message consumers (Batch processing with reactive programming)
  - Spreadsheets
  - Abstraction over Asynchronous Processing
    - Abstract whether or not your program is synchronous or asynchronous
  - Focuses on streams of data, but not for every thing CRUD applications are still alive and well

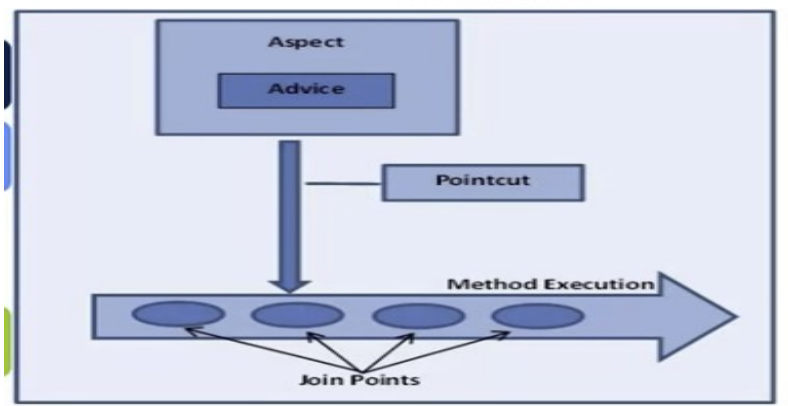
What is reactive programming

- Core Features
  - Data Streams
    - Can be just about anything from mouse streams to text (Mouse Clicks, JMS Messages, Rest Messages, Twitter Feed)
    - A Stream is a sequence of event ordered in time
    - Event you want to listen to
  - Asynchronous
    - Events are captured asynchronously
    - function is defined to execute when an event is emitted and another function is defined if an error emitted and another for complete
  - Non-blocking
    - Will process available data ask to be notified when more data is available
    - NodeJS setup an event loop (Non Blocking) and if you block you kill the system performance
    - Like a callable future where it reaches the processing. The thread will stay longer on the CPU which it handles more effectively
  - Backpressure
    - Throttle Data – The ability for the subscriber to throttle data
  - Failure as messages
    - Exceptions will not be thrown in a traditional sense which would break the processing of a data streams, but processed by a handler funciton
  - Gang of Four Observer Pattern
    - Items are emitted by the observable then a transformation is applied and the result is put back on the stream

AOP

- Goals of AOP
  - Aims to increase modularity by allowing the separation of cross cutting concerns
  - Cross Cutting concerns are aspects of a program which affect other concerns
    - want the code in one place
  - Example of concerns : Logging, Security, Transactions
  - Concerns often cloud the code
- Spring AOP is Proxy Based
- AOP Components
  - Aspect Unit of modularity for cross-cutting concerns
  - JoinPoint Well defined points in the program flow (application)
  - PointCut Queries where advice executes – Associates the joint point with the advice
  - Advice Block of code that runs based on the point cut definition
  - Weaving Done at run or compile time, inserts advice (cross cutting concerns) into the code (core concerns)
    - compile Time – Aspect J
    - run time – Spring AOP





AOP

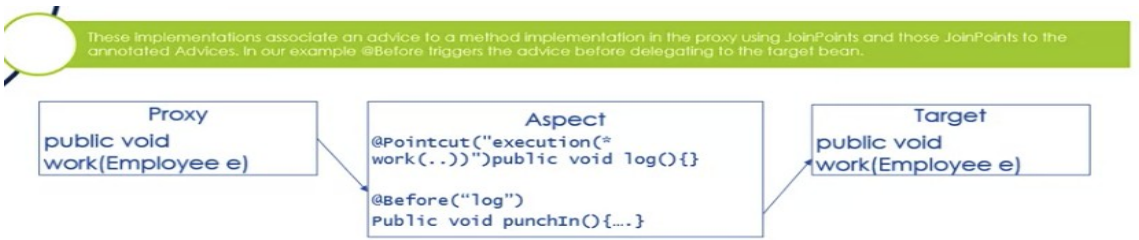
- Spring AOP
  - Framework that builds on the aop alliance interfaces
  - Aspects are coded with pure java code
  - Objects obtained from the IoC container can be transparently advice based on configuration
  - Has build in aspects providing transaction management, performance monitoring and Security
- Enable AOP
  - Based off a framework called AspectJ and is a subset of AspectJ
  - AspectJ compiles new class files with the advice weaved in
  - Spring AOP builds a proxy object surrounding the invocation with the advice
  - Configure Spring Context for Aspect J
    - add @EnableAspectJAutoProxy
    - For a spring boot project
      - <dependency>
        - <groupId>org.springframework.boot</gorupId>
        - <artifactId>spring-boot-starter-aop</artifactId>
      - </dependency>
      - @Configuration
      - @EnableAspectJAutoProxy
      - public class AppConfig

AOP

- Aspects and Join Points
  - @Aspect indicates to Spring that the class is an aspect
    - It must be Spring managed
  - A point cut is a label that effectively points to an expression that defines an entry point into the application I.e (A method execution)
    - defined by the @Pointcut annotation and an express in the AspectJ Language to specify the joinpiont
    - @Pointcut is applied to a method which becomes the name of the point cut (method signature should be public void)
      - AspectJ specifies which joinpoints are of interest
  - Example
    - import org.aspectj.lang.annotatoin.Aspect
    - @Aspect
    - @Named
    - public class EmployeeAspect {
      - @Pointcut(“execution(* work (..))”)
      - public void log() { }
    - }

Advice

- An advice is associated with a point cut that points to an expression that defines an entry point into the application
  - Advice is specified by an annotation (there are several) on a method.
  - point cut name specifying when the advice is to run
  - The annotated method defines the action of the advice
- Example
  - @Aspect
  - public class WorkSupport {
    - public void log() {}
    - @Before(“log()”) public void punchIn() { System.out.println(“About to clocked in a t” + new Date().toString()); } }



- AOP Using Proxies
  - start up time of the Spring Boot Application, Aspects, are instantiated early as the Spring Container is inflated
  - As more beans are created by the container, the container used introspection to match the method signatures to the JoinPoints in the Aspect
    - If there is a match the container conceptually wraps the target bean in a proxy who has to implement the same methods as the target bean
    - Implementations associate an advice to a method implementations in the proxy using JointPoints and those JoinPoints to the annotated Advice

Getting Information About the Arguments to a method

- Information about the JointPoint
  - inject in a Joint point instance
  - provides information
    - about signature
    - parameters
  - cannot change the arguemnts

In our example it does not print out who was clocking in. We need to change our Advice

```
@Aspect
public class WorkSupport {
    @Pointcut("execution(* work(..))")
    public void log(){}
    @Before("log()")
    public void punchIn(Joinpoint jp) {
        System.out.println("About to clocked in at " + new Date().toString());
        if (jp.getArgs()[0] instanceof Employee){
            Employee employee = (Employee) jp.getArgs()[0];
            System.out.println(employee.getFirstName() + " clocked in at " + new
              Date().toString());
        }
        System.put.println(jp.getSignature().getName());
    }
}
```

More Advice

- Before
  - Call before method execution
- After
  - Always called after target execution
- After Returning
  - Called after normal method execution without exception
  - Have two parameters can get at the variable foo to read it.
  - @AfterReturning will only be triggered on normal return from the target bean, @After always executes even if the target bean method threw an Exception. It also provides a means to get hold of the returned object form the target bean.
- After throwing
  - Called After Method execution exits with exception
  - Does not catch but can throw another exception if desired automatically rethrows original exception
- around
  - Run the advice before and after the advised method is invoked
- Spring supports the AspectJ annotation style and schema-based approach

```
@After("log()")
public void after(JoinPoint jp) {
    LOG.info("Invoked Method After->" +
    jp.getSignature().getName());
}
@AfterReturning(pointcut="log()", returning =
"foo")
public void afterNormalReturn(JoinPoint jp,
String foo) {
    LOG.info("we got back " + foo);
}
@AfterThrowing(pointcut="log()",throwing = "e")
public void catchAndthrow(JoinPoint jp,
Exception e) {
    LOG.info("we got back " + e);
    throw new RuntimeException("Contact
    Support");
    }
}
```

Example

```
@Aspect
@Named
public class MyAspect {
private Logger LOG = LoggerFactory.getLogger(this.getClass().getSimpleName());
  @Pointcut ("execution(* com.test.service.*.*(..))")
public void log() {}
  @Before (“ log() “)

public void before( JoinPoint jp) {
   LOG.info("Invoked Method Before->" + jp.getSignature().getName());
}
```

Example 2

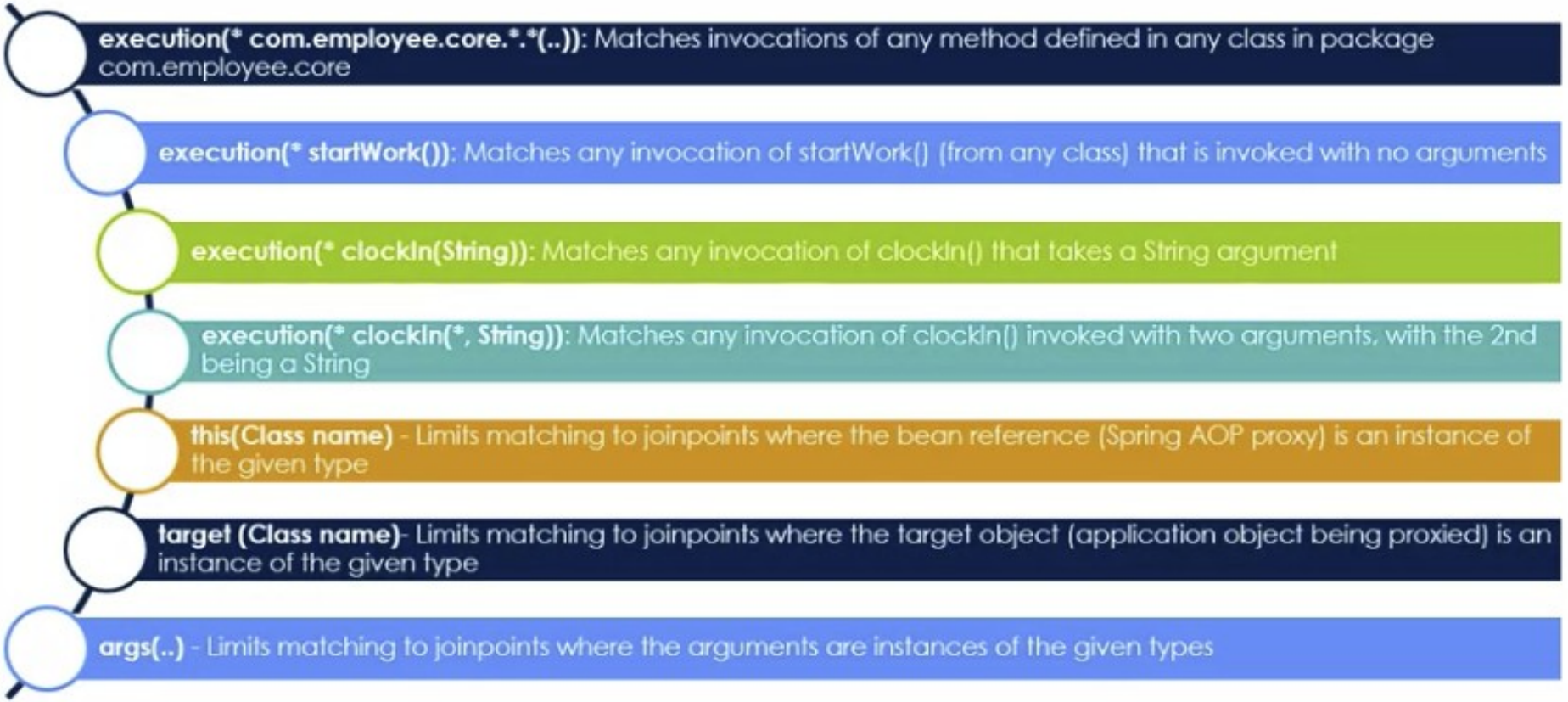
```
@Aspect
@Named
public class MyAspect {
private Logger LOG = LoggerFactory.getLogger(this.getClass().getSimpleName());
  @Pointcut ("execution(* com.test.service.*.*(..))")
public void log() {}
 @Before    (“ log()  “)

public void before( JoinPoint     jp) {
   LOG.info("Invoked Method Before->" + jp.getSignature().getName());
}
```

AOP

- Expressions
  - JoinPoint expressions are express as AspectJ expressions
  - designator that specify the joinpoint
  - execution designator
    - The return type, name and parameter are required
    - All other are optional
    - Wildcard are allot in the patterns
  - Example
    - execution(modifiers-pattern>? ret-type-pattern declaring-type-pattern? name-pattern(param-pattern) throws-pattern?)
      - public String com.employee.ServiceBean.methodName(String) Exception
  -

AOP


execution(* com.employee.core.*.*(..)): Matches invocations of any method defined in any class in package com.employee.core
execution(* startWork()): Matches any invocation of startWork() (from any class) that is invoked with no arguments
execution(* clockIn(String)): Matches any invocation of clockIn() that takes a String argument
execution(* clockIn(*, String)): Matches any invocation of clockIn() invoked with two arguments, with the 2nd being a String
this(Class name) - Limits matching to joinpoints where the bean reference (Spring AOP proxy) is an instance of the given type
target (Class name)- Limits matching to joinpoints where the target object (application object being proxied) is an instance of the given type
args(..) - Limits matching to joinpoints where the arguments are instances of the given types

AOP

- Around
  - The advice method must take a ProceedingJoinPoint parameter.
    - Code after the call to proceed() will execute after the jointpoint
    - takes the preceding join point will give used the metadata of the joinpoint before
      - proxy change – Multiple around advices wrapping a target off various expression – Join Point Cuts
      - Using the preceding join point can transverse down the proxy chain using proceed funciton()
  - Example
    -

```
@Around("log()")
public Object goToandFromWork(ProceedingJoinPoint joinPoint) throws Throwable
{
   System.out.println("Starting the day at " + new Date().toString());
   Object object =joinPoint.proceed();
   System.out.println("Leaving for the day at " + new Date().toString());
   return object;
}
```

AOP

Replace out bound response types

With a little extra help from the joinpoint we can get at the input arguments and the return objects

```
@Around("add() && args(employee)")
public Object goToandFromWorkWithArgs(ProceedingJoinPoint joinPoint ,Employee
employee) throws Throwable {
    System.out.println("Hello " + employee.getFirstName() + " " +
    employee.getLastName());
    System.out.println("Your First Day " + new Date().toString());
    Employee empl = new Employee();
    empl.setFirstName("Javert");
    empl.setLastName("Valgent");
    empl.setDeptId(5);
    Object object =joinPoint.proceed(new Object[]{empl});
    System.out.println("Everything OK " + object);
    System.out.println("Leaving for the day at " + new Date().toString());
    return false;
}
```

The around advice provides sucurity and transactional advisors in app