Spring Framework

Introduction to Spring - 4 Day

Overview

- Introductions
  - Instructor - Geoff Matrangola - geoff@matrangola.com @triglm
  - Company - DevelopIntelligence http://www.developintelligence.com/ (show 2 slides)
  - Students
    - Names, Current projects, Class Expectations
    - Mention Pre-Assessment
      - Strong Java, SQL, JDBC
      - Need Spring, NoSQL and Cassandra
      - Ask about JPA, JUnit/testing
  - Course - How to develop a Rest API Using Spring
- Logistics
  - Start, end, break times
  - Facilities
- Class Agenda
  - See Class outline
- Class Flow
  - Slides
  - Demo
  - Lab

What is the Spring Framework?

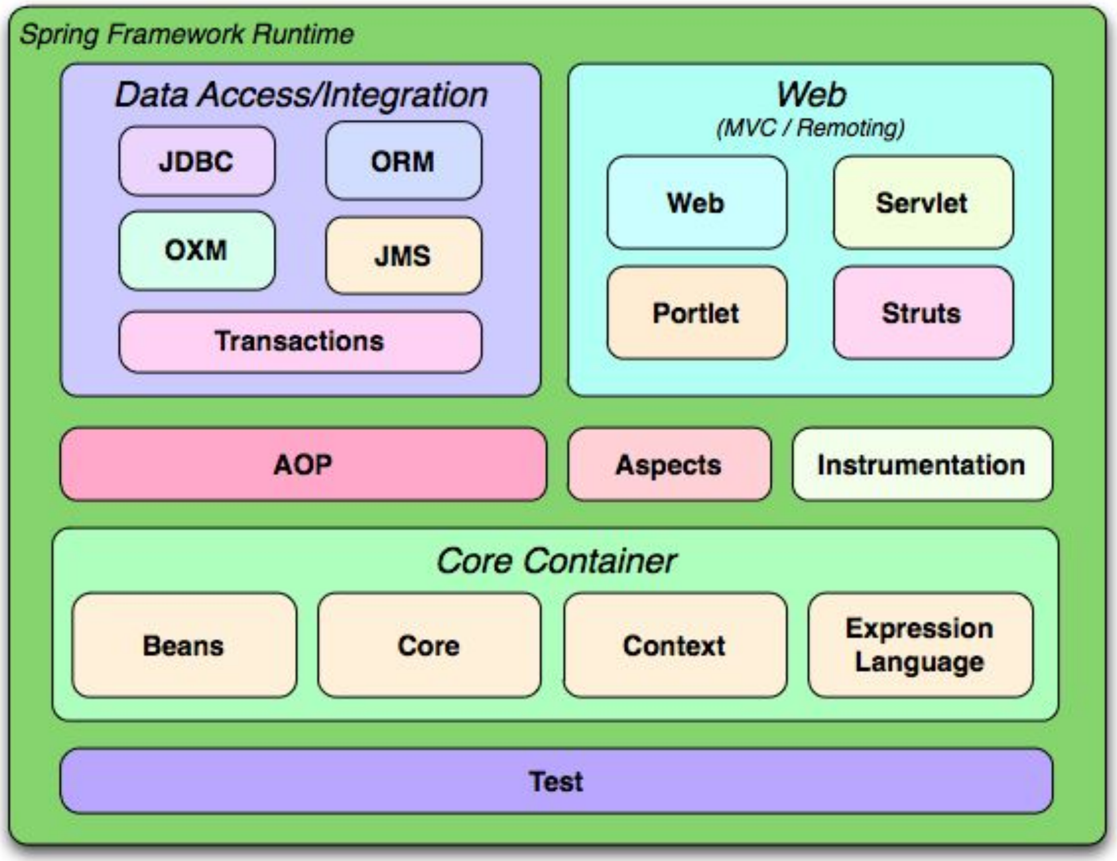
- Java based framework applications
- Library agnostic
- Configuration by convention and automation
- Java Annotations and/or XML Configuration
- Practically any web server with servlet compatibility
- Dependency Injection and Inversion of Control

What is the Spring Boot?

- Java based framework for stand-alone applications
- Rich set of libraries that can be integrated into your application
- Opinionated starter libraries (Maven Repos)
- Configuration by convention and automation
- Java Annotations
- Embedded Web Server (Tomcat by default)
- Easy Database configuration (JPA implementation by default)
- Dependency Injection and Inversion of Control

Key Elements of the Spring Framework

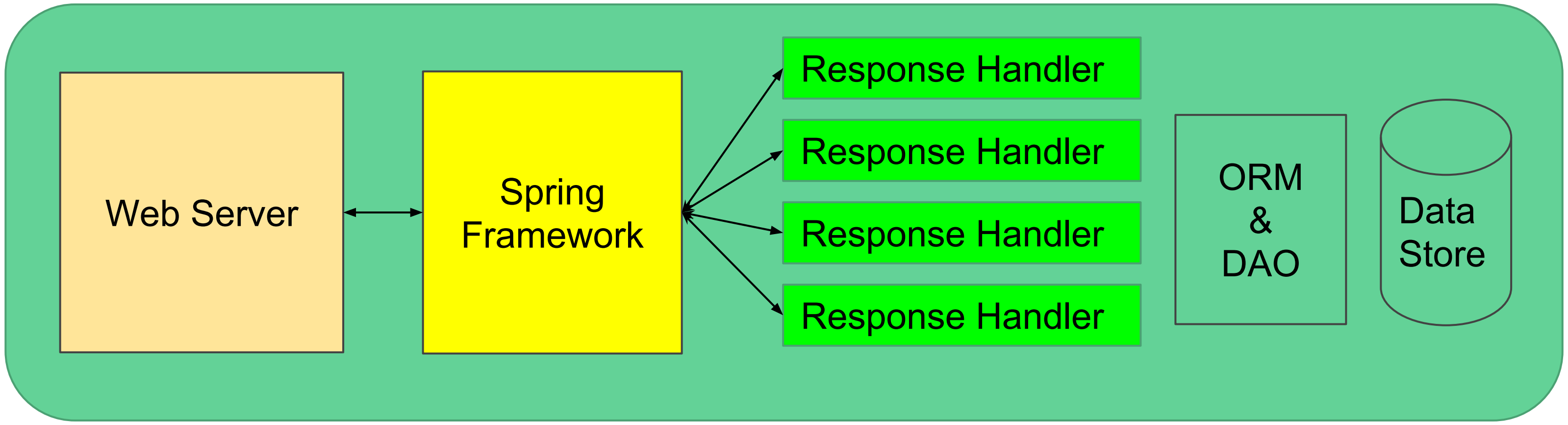
- Modules
- Core Container
- Beans
- Context
- AOP
- Other - Data, Web, Test, Instrumentation



https://docs.spring.io/spring/docs/3.0.x/spring-framework-reference

Inversion of Control (IoC)

- The framework maintains the flow of execution & setting object dependencies
- You wire in the custom business routines
- You define the objects
- You are provided objects with all their properties wired up.
- Request protocol handled by Spring and the Web Server- you write the response handler



Dependency Injection

- Objects define their dependencies ONLY
  - Constructor Arguments
  - Factory Method Arguments
  - Properties, set by Factory Method
- The container *injects* the *dependencies* when it creates the object instance
- Objects that are managed in this way are called **Spring Beans**
- **Spring Beans** are instantiated, and managed by the Spring IoC Container.

https://docs.spring.io/spring/docs/3.0.0.M3/reference/html/ch04s03.html

Spring Bean Scope

Scope	Description
singleton	Scopes a single bean definition to a single object instance per Spring IoC container.
prototype	Scopes a single bean definition to any number of object instances.
request	Scopes a single bean definition to the lifecycle of a single HTTP request; that is each and every HTTP request will have its own instance of a bean created off the back of a single bean definition. Only valid in the context of a web-aware SpringApplicationContext.
session	Scopes a single bean definition to the lifecycle of a HTTP Session. Only valid in the context of a web-aware SpringApplicationContext.
global session	Scopes a single bean definition to the lifecycle of a global HTTP Session. Typically only valid when used in a portlet context. Only valid in the context of a web-aware Spring ApplicationContext.

Demo/Lab 1
Setup & RestController

Demo/Lab 1: Hello World REST Web Service

- Simple lab to verify your configuration
- Using Spring Initializer to build base project
- Incremental development to bring explore concepts of the Spring Boot throughout the entire class.
- REST service responds with JSON
- Intellij, Gradle, Spring Boot, Tomcat, etc.

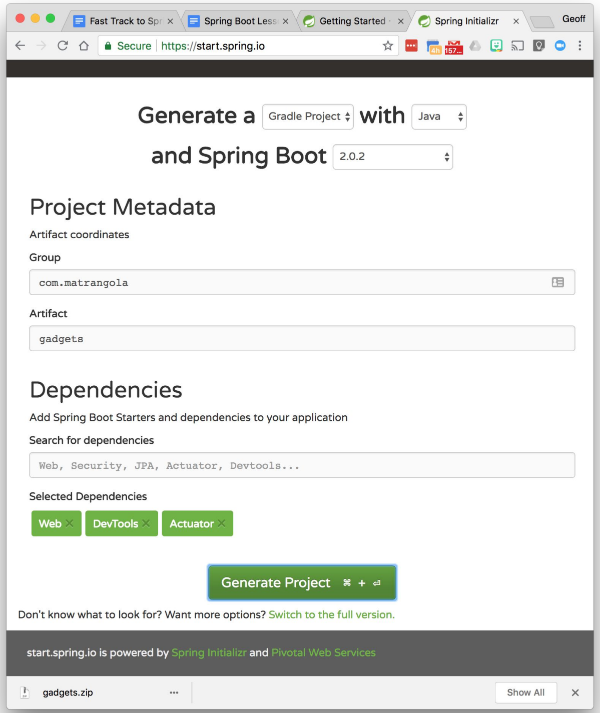
Setup

- Intellij Idea 2018.1.2
- Java JDK 8
- Chrome Web Browser
- MySQL
- Postman to verify REST
- Internet Access

Spring Initializr

https://start.spring.io/

- Gradle Project
- Java
- 2.0.2
- Group: com.whatever
- Artifact: gadgets
- Dependencies: Web, Actuator, DevTools
- Download
- Unzip



Generate a Gradle Project with Java and Spring Boot 2.0.2

Project Metadata

Artifact coordinates

Group

com.matrangola

Artifact

gadgets

Dependencies

Add Spring Boot Starters and dependencies to your application

Search for dependencies

Web, Security, JPA, Actuator, Devtools...

Selected Dependencies

Web ✕ DevTools ✕ Actuator ✕

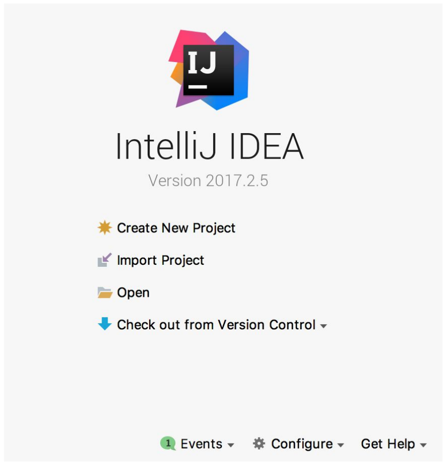
Generate Project ⌘ + ⏎

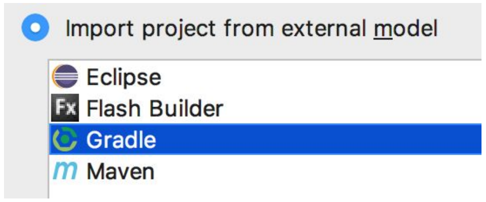
Don't know what to look for? Want more options? Switch to the full version.

start.spring.io is powered by Spring Initializr and Pivotal Web Services

Import Part 1

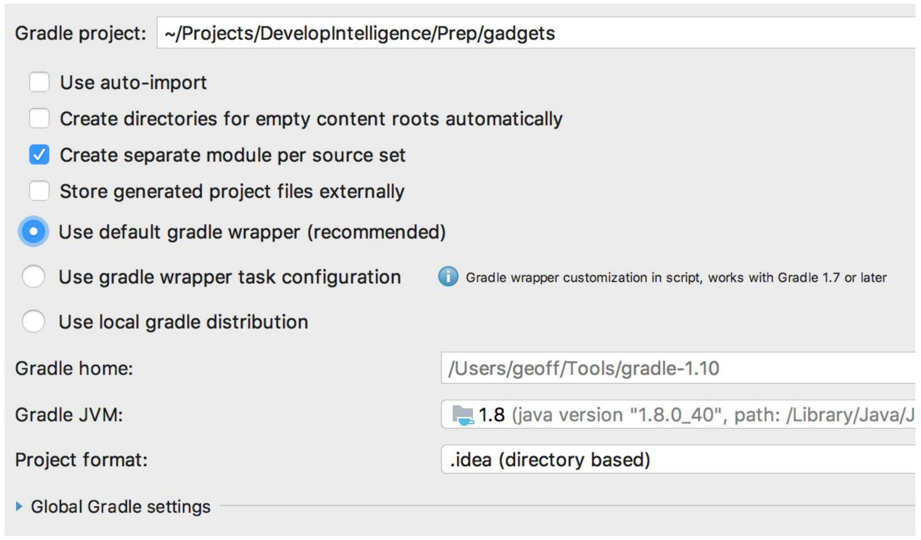
- Launch IntelliJ Idea
- Import Project
- Select Downloaded & Unzipped Directory
- Select Import project...
- Gradle





Import Part 2

- Gradle project: ~/your/project/dir
- Create separate module…
- Use default gradle wrapper
- Finish



Gradle project: ~/Projects/DevelopIntelligence/Prep/gadgets

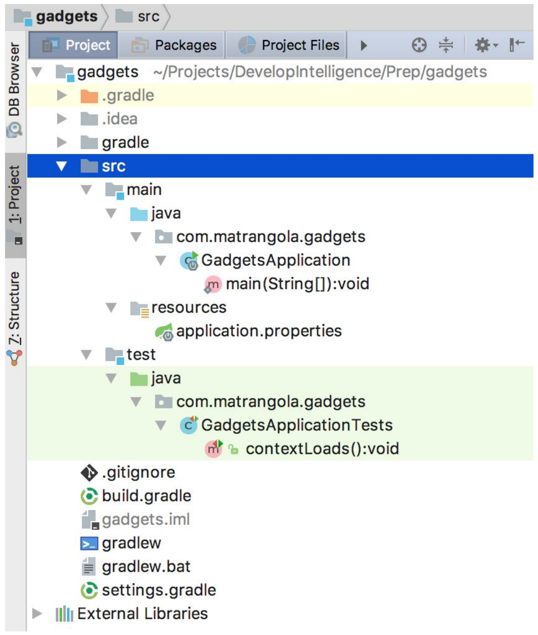
- [ ] Use auto-import
- [ ] Create directories for empty content roots automatically
- [✓] Create separate module per source set
- [ ] Store generated project files externally

[✓ Use default gradle wrapper (recommended)]
Use gradle wrapper task configuration — Gradle wrapper customization in script, works with Gradle 1.7 or later
Use local gradle distribution

Gradle home: /Users/geoff/Tools/gradle-1.10
Gradle JVM: 1.8 (java version "1.8.0_40", path: /Library/Java/J
Project format: .idea (directory based)

Project Structure

- .idea - IDE stuff
- gradle - automated build stuff
- src - Java and Resources
- build.gradle - build configuration
- Other files



```
gadgets  ~/Projects/DevelopIntelligence/Prep/gadgets
  .gradle
  .idea
  gradle
  src
    main
      java
        com.matrangola.gadgets
          GadgetsApplication
            main(String[]):void
      resources
        application.properties
    test
      java
        com.matrangola.gadgets
          GadgetsApplicationTests
            contextLoads():void
  .gitignore
  build.gradle
  gadgets.iml
  gradlew
  gradlew.bat
  settings.gradle
External Libraries
```

Annotations Used in Demo

@RestController - Identify the Rest Controller for the Framework

@RequestMapping - Path of the URL mapped from the web server to the code

@RequestParam - Request params in the URL mapped to method parameters

Lab 1 - Starter

```java
package com.matrangola.gadgets.data.model;

public class Customer {
  private String firstName;
  private String lastName;

  public String getFirstName() {
     return firstName;
  }

  public void setFirstName(String firstName) {
     this.firstName = firstName;
  }

  public String getLastName() {
     return lastName;
  }

  public void setLastName(String lastName) {
     this.lastName = lastName;
  }
}
```

```java
@RestController
public class CustomerController {
  @RequestMapping("/makeCustomer")
  public Customer makeCustomer(
                        @RequestParam(value="last") String lastName,
                        @RequestParam(value="first") String firstName) {
     Customer customer = new Customer();
     customer.setFirstName(firstName);
     customer.setLastName(lastName);
     return customer;
  }
}
```

Lab 1

https://github.com/gmatrangola/indicator

1. Specify and download Spring Initializer
2. Unzip
3. Import into IDE
4. Use annotations to create ResetController
5. Start Web Server & Application
6. Test with HTTP Request, curl or similar
7. Add Email Address to Customer class
8. Add optional email address param to /makeCustomer

Data Management

Spring Persistence with Cassandra NOSQL

Setup

1. `docker pull cassandra:latest`
2. `docker run --name spring-cassandra -p 7000-7001:7000-7001 -p 7199:7199 -p 9042:9042 -p 9160:9160 -d cassandra:latest`
3. `Docker start spring-cassandra`

Cassandra cqlsh

```
docker exec -ti spring-cassandra cqlsh localhost
```

Data Management

- Entity Domain - Table mapping
  - Defines Primary Key
  - Fields
  - Indexing
  - Maps to Cassandra Tables
- Repository
  - CassandraRepository
  - Interfaces that can be used by services to access Entities in the Data Store
  - Interfaces with Cassandra Cluster

Persistence - Some Annotations used

@Table - Define the Table where the Entity is stored

@PrimaryKey - Field is used as a column

@CassandraType - Details about Primary Key

Cassandra Database

- Update build.gradle to include Cassandra and dropwizard metrics
- Create config package
- Create CassandraConfig.java
- Create data.repository package
- Create CustomerRepository.java
- Add Annotations to Customer.java

Data Management Demo Results: Need Config

```
2018-11-11 14:13:29.072  INFO 811 --- [  restartedMain] o.apache.catalina.core.StandardService   : Stopping service [Tomcat]
2018-11-11 14:13:29.087  INFO 811 --- [  restartedMain] ConditionEvaluationReportLoggingListener :

Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2018-11-11 14:13:29.105 ERROR 811 --- [  restartedMain] o.s.boot.SpringApplication               : Application run failed

org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'customerController': Unsatisfied
dependency expressed through field 'customerService'; nested exception is
org.springframework.beans.factory.UnsatisfiedDependencyException: Error creating bean with name 'customerServiceImpl': Unsatisfied
dependency expressed through field 'customerRepository'; nested exception is
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'customerRepository': Cannot resolve
reference to bean 'cassandraTemplate' while setting bean property 'cassandraTemplate'; nested exception is
org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'cassandraTemplate' defined in class path
resource [com/matrangola/msci/config/CassandraConfig.class]: Bean instantiation via factory method failed; nested exception is
org.springframework.beans.BeanInstantiationException: Failed to instantiate
[org.springframework.data.cassandra.core.CassandraAdminTemplate]: Factory method 'cassandraTemplate' threw exception; nested
exception is org.springframework.beans.factory.BeanCreationException: Error creating bean with name 'sessionFactory' defined in class
path resource [com/matrangola/msci/config/CassandraConfig.class]: Bean instantiation via factory method failed; nested exception is
org.springframework.beans.BeanInstantiationException: Failed to instantiate [org.springframework.data.cassandra.SessionFactory]:
Factory method 'sessionFactory' threw exception; nested exception is org.springframework.beans.factory.BeanCreationException: Error
creating bean with name 'session' defined in class path resource [com/matrangola/msci/config/CassandraConfig.class]: Invocation of init
method failed; nested exception is com.datastax.driver.core.exceptions.NoHostAvailableException: All host(s) tried for query failed (tried:
localhost/127.0.0.1:9042 (com.datastax.driver.core.exceptions.TransportException: [localhost/127.0.0.1:9042] Cannot connect),
localhost/0:0:0:0:0:0:0:1:9042 (com.datastax.driver.core.exceptions.TransportException: [localhost/0:0:0:0:0:0:0:1:9042] Cannot connect))
	at
org.springframework.beans.factory.annotation.AutowiredAnnotationBeanPostProcessor$AutowiredFieldElement.inject(AutowiredAnnotati
onBeanPostProcessor.java:596) ~[spring-beans-5.1.2.RELEASE.jar:5.1.2.RELEASE]
	at org.springframework.beans.factory.annotation.InjectionMetadata.inject(InjectionMetadata.java:90)
~[spring-beans-5.1.2.RELEASE.jar:5.1.2.RELEASE]
```

Live Demo 3b: Setup Cassandra

```
docker start spring-cassandra
```

```
docker exec -ti spring-cassandra cqlsh localhost
```

```
CREATE KEYSPACE IF NOT EXISTS msci WITH replication = {'class':'SimpleStrategy', 'replication_factor':1};

create table msci.customer (id UUID, firstName varchar, lastName varchar, primary key(id));
```

```
#
# Cassandra Config
#
spring.data.cassandra.keyspace-name=msci
spring.data.cassandra.contact-points=localhost
spring.data.cassandra.prot=9042
```

Demo 3c - Access Repository from Controller

- Create RequestMapping for “/new” to addCustomer
- Create RequestMapping for “/” to get all Customers
- Run
- Verify in CQL

Live Demo: Select Table

```
qlsh:msci> use msci;
cqlsh:msci> select * from customer;

 id                                   | firstname | lastname
--------------------------------------+-----------+------------
 821ddb91-444e-48e1-8d52-c2a4ecf0fc21 |     Geoff | Matrangola

(1 rows)
cqlsh:msci>
```

Lab 3: Data Management

1. Create CassandraConfig
2. Add Cassandra Annotations to Customer Class
3. Create CustomerRepository Interface
4. Setup Cassandra database
5. Create applicaiton.properties
6. CustomerContoller
   a. Add @Autowire for CustomerRepository
   b. Run HTTP Get on /makeCustomer
   c. Add getAll() the get all customers with RequestMapping /
   d. Add get(id) to get a customer by ID. *hint*: customerRepository.findById(id)
7. Verify output in CQL

Spring Configuration Management

resources directory

- application.properties
- logback.xml (depending on logging solution)

Rest and Test

REST

Representational STate Transfer

URI as User Interface

https://myserver.com/myapp/users/bob/birthday

HTTP Verbs

- GET - Request a resource
- DELETE - Remove a resource
- PUT - Upload a resource
- POST - Do something with the uploaded resource, may be handled same as PUT

OpenWeatherMap

https://openweathermap.org/current

http://api.openweathermap.org/data/2.5/weather?zip=02451,us&units=imperial&appid=4d36b5f1fce463fe1647b8b9711bf707

@RestController

@RestController = @Controller + @ResponseBody

Specialized @Component detected through Classpath Scan at startup.

@Controller - Defines a Web Controller that the framework will scan for @RequestMappings to handle IoC request mappings from the web server.

@ResponseBody - Indicates that values returned from methods should be sent as the HTTP Response. Default converts to JSON.

```
@RestController
public class CustomerController {
  //...
}
```

@RequestMapping

```
@RestController
@RequestMapping(value = "/customers", produces = {"application/json"})
public class CustomerController {
        @RequestMapping(path = "/makeCustomer", method = RequestMethod.GET)
        public Customer greeting(
                @RequestParam(value="last") String lastName,
                @RequestParam(value="first") String firstName) // ...
```

Connects the URI to the correct method

- Valid at Class and Method level
- Options
    - path (default) - Path part of the URL mapped to this controller
    - value - for servlet mapping (i.e. “/myPaath.do”, “/myPath/*.do”)
    - method - GET, HEAD, POST, PUT, PATCH, DELETE, OPTIONS, TRACE
    - params - list of parms and values to map the correct method (i.e. params = {“foo=100”})
    - headers - list of header values to match (i.e. headers = {“content-type=text/plain”, “content-type=application/json”})
    - consumes - list the types that this method consumes (i.e. consumes = {“application/json”, “application/xml”})
    - produces - list the types that will be produced (i.e. produces = {“application/json”)

Examples: https://springframework.guru/spring-requestmapping-annotation/

@RequestParam and @PathVariable

- annotate parameters of a method that match Query Strings or parts of the path
- name - name of the query string or {pathVariable}
- required - default *true*
- default - default string

```
@RequestMapping(value = "/new", method = RequestMethod.GET)
public Customer add(@RequestParam(value="last") String lastName,
            @RequestParam(value="first") String firstName,
            @RequestParam(name = "birthday", required = false) String birthdayText) // ...


@RequestMapping("/older/{age}")
public List<Customer> older(@PathVariable int age)  //...
```

@RequestBody

- Indicates parameter is the body of the HTTP Request
- required - default *true*

```
@RequestMapping(value = "/new", method = RequestMethod.PUT)
public Customer add(@RequestBody Customer customer) {
// ...
```

Demo/Lab 4: RequestMapping

Demo 4 - Request Mapping

- Modify Customer to add birthday so that we can have more data to play with
- Add a getCustomers() that passes through the CustomerRepository to get a list of customers to satisfy Customer requests.
- Add class level RequestMapping for CustomerController
- Change “/makeCustomer” to “/new”, add RequestType.GET
- Add another /add that takes a Customer as a @RequestBody parameter and has a RequestType.PUT
- Make foo() with crazy RequestMapping value strings for wildcards
- Create test customers with http request

Lab 4: RequestMapping

1. Implement the code from the Demo in your application.
2. Add an optional age query string parameter to add (/new) that takes a String and uses SimpleDateFormat to convert it to a java.util.Date.
3. Set the Date in the Customer object
4. For now, catch and swallow the exception from SimpleDateFormat.parse(). We'll cover exception handling soon
5. Create a request mapping to get a customer by ID (DB PK) using a @PathVariable
6. Add another /add that takes a Customer as a @RequestBody parameter and has a RequestType.PUT
7. Add @RequestMapping to the all customers when request is "/customers/"
8. Add a top level / that gets all the customers
9. Add email and zipcode to the Customer domain class bean

Lab 4: Solution Part 1

```
private static final SimpleDateFormat BIRTHDAY_TEXT_FORMAT = new SimpleDateFormat("YYYYMMdd");
@RequestMapping(value = "/new", method = RequestMethod.GET)
public Customer add(@RequestParam(value="last") String lastName,
            @RequestParam(value="first") String firstName,
            @RequestParam(name = "birthday", required = false) String birthdayText) {
  Customer customer = new Customer();
  customer.setFirstName(firstName);
  customer.setLastName(lastName);
  if(birthdayText != null) {
     try {
        customer.setBirthday(BIRTHDAY_TEXT_FORMAT.parse(birthdayText));
     } catch (ParseException e) {
        e.printStackTrace();
     }
  }
  customerService.addCustomer(customer);
  return customer;
}
```

```
@RequestMapping(path = "/picture/{customerId}", method = RequestMethod.PUT, consumes = {"image/jpeg"})
public String picture(@PathVariable("customerId")int customerId, @RequestBody byte[] bytes) {

  return "Customer ID: " + customerId + " uploaded " + bytes.length + " bytes";
}
```

Lab 4 Solution Part 2

```
@RequestMapping("/{id}")
public Customer getById(@PathVariable("id") UUID id) {
   return customerService.getCustomer(id);
}
```

```
@Override
public Customer getCustomer(UUID id) {
   return customerRepository.findById(id).get();
}
```

REST API Standards Help maintain your users sanity

REST API Standards Suggestions

- Organize logical URL Hierarchy
- Organize and name controller classes to match the Request Mapping paths as closely as possible
- Be consistent with capitalization and Query String Parameter names
- Name methods to match Request Mappings as closely as possible
- Always specify RequestMethod
- Always specify consumes
- Try to make GET read only
- Avoid using GET with @RequestParam to modify data
- Use PUT to insert and POST to modify (when practical)
- Specify “path” vs “value” in @RequestMapping

Demo 5a: URL Path Variables

Refactor

1. path = “/customers”, consumes and produces default for class
2. value -> path
3. Add RequestMethod
4. Make zipcode RequestMapping that returns Customers with zip - Inefficient use of Database with Java Streams for all values
5. Use Java streams to get value (bad performance)

Demo 5b: Mapping Path Variable to Cassandra Index

Refactor

1. Make zipcode RequestMapping more efficient
2. Annotate zipcode as @Indexed
3. Implement Repository and Service methods for zipcode
4. Use the new service method in the Controller method

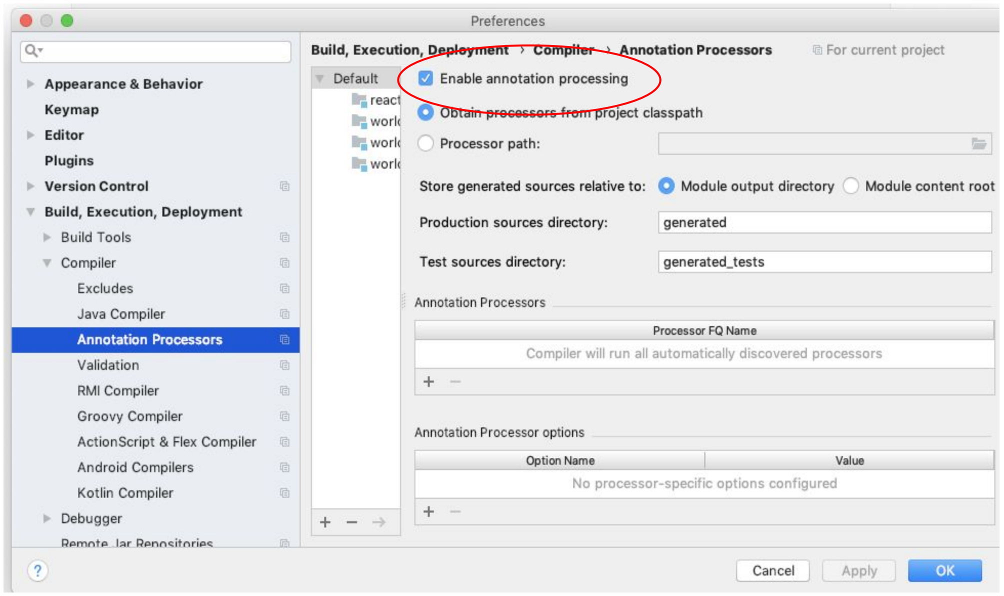
Lab 5: Lookup by index

1. Clean up naming of RequestMappings in your CustomerController
2. Annotate zipcode and email with @Indexed
3. Create Service and Repository methods to for efficient lookup by zipcode and email
4. Lookup by zipcode should return a list
5. Lookup by email should return a single Customer
6. Add zipcode path variable to CustomerController
7. Add email RequestMapping to CustomerController

Lombok

Improved Data modeling with Lombok

- Frequently Used in Spring Projects
- Reduces Boilerplate Java Code
- Minimalist Web Page: https://projectlombok.org/
- Getter/Setter - Real vs. Template
- Constructors
- Logging
- Add to Gradle Dependancy: `compileOnly('org.projectlombok:lombok')`
- Update IDE Settings


Preferences

Appearance & Behavior
Keymap
Editor
Plugins
Version Control
Build, Execution, Deployment
Build Tools
Compiler
Excludes
Java Compiler
Annotation Processors
Validation
RMI Compiler
Groovy Compiler
ActionScript & Flex Compiler
Android Compilers
Kotlin Compiler
Debugger
Remote Jar Repositories

Build, Execution, Deployment › Compiler › Annotation Processors   For current project

Default
react
worl
worl
worl

[✓] Enable annotation processing
(●) Obtain processors from project classpath
( ) Processor path:

Store generated sources relative to: (●) Module output directory ( ) Module content root
Production sources directory: generated
Test sources directory: generated_tests

Annotation Processors

Processor FQ Name
Compiler will run all automatically discovered processors

Annotation Processor options

Option Name	Value
No processor-specific options configured	

Cancel   Apply   OK

Quick Lombok Demo

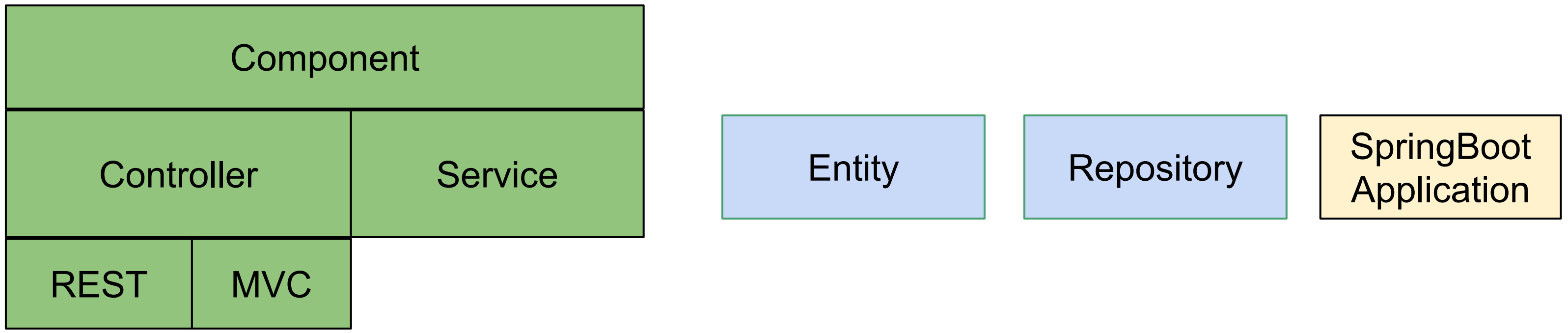
- Import into Gradle
- Set IDE Preference
- Update Customer Data
- Rebuild

Core Spring Boot Components and Classes

Services

- Testable Business Logic
- @Service
- @Component
- Stateless
- @Service vs Microservices
- @AutoWired clients
- User interface for testing

Core Spring Boot Components and Classes



Components

- Found with Spring Boot Classpath Search
- Controller = service with Presentation (REST API or MVC Web)
- RestController is a Controller with a Response Body
- Service is stand-alone business logic

Component Annotations

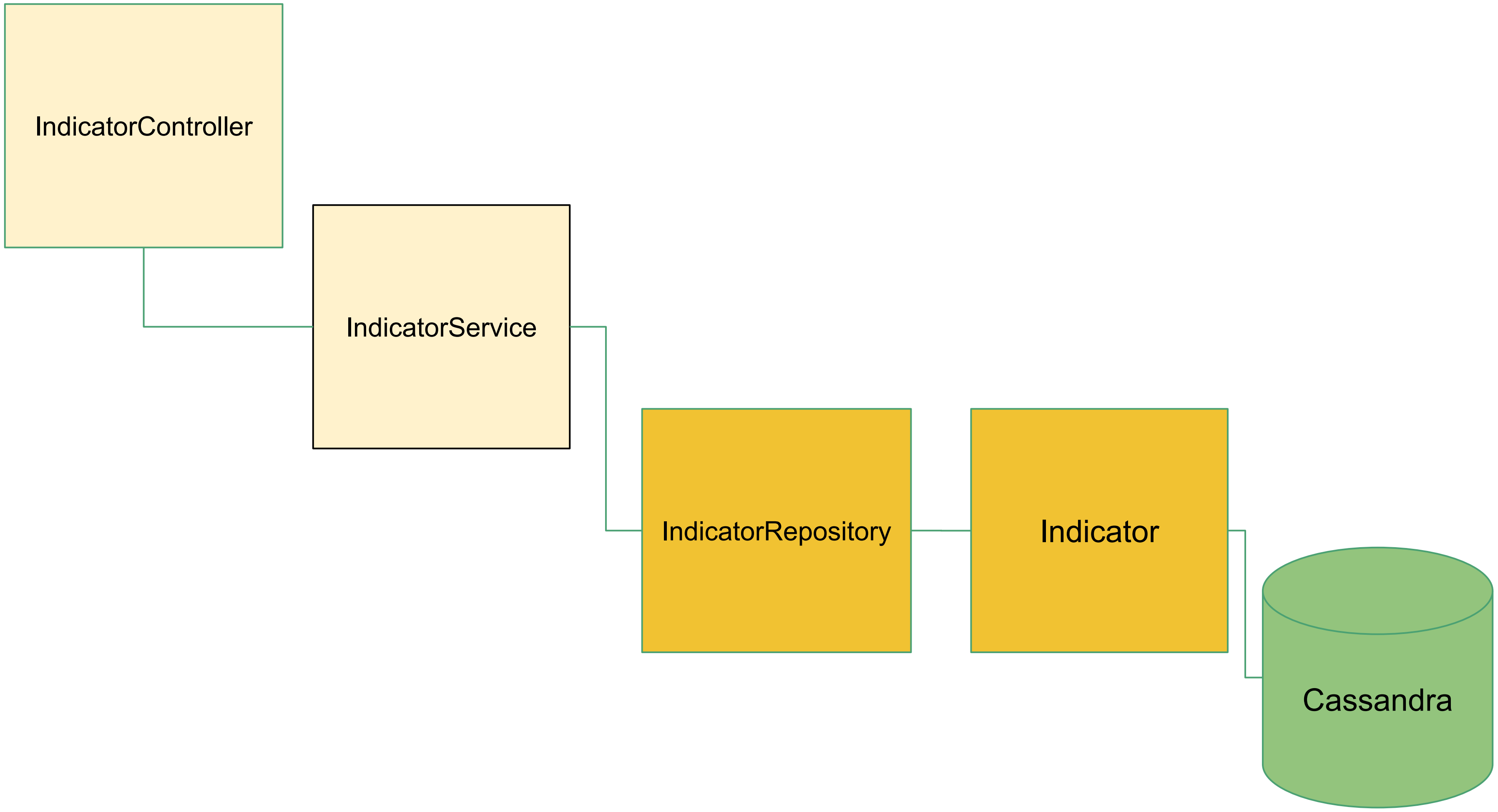
On the Service Class

@Service - Class is a component

In the “Client” class

@Autowired - marks automatically referenced component using Spring’s dependency injection.

Indicator Summary



Demo 6: Services

- Create Indicator Model Class
  - Annotate for Cassendra
- Run Webservice to automatically create the Table
- Import CSV file
- Create IndicatorRepository.java
  - Create Stream<Indicator> findAllByIndicator(String code)
- Create new `service` package
- Create IndicatorService Interface
- Create IndicatorServiceImpl Class
  - Extend IndicatorService and add Annotations
  - Autowire IndicatorRepository
  - Create worldWideAverage(String code) to compute average for 2017 for code
- Create IndicatorController
  - Autowire IndicatorService
  - Add @RequestMapping @RestController
  - Create avg to return the average from IndicatorService.worldWideAverage()

Lab 6 Services

- Create Indicator Model Class
  - Annotate for Cassendra
- Run Webservice to automatically create the Table
- Import CSV file
- Create IndicatorRepository.java
  - Create Stream<Indicator> findAllByIndicator(String code)
- Create new `service` package
- Create IndicatorService Interface
- Create IndicatorServiceImpl Class
  - Extend IndicatorService and add Annotations
  - Autowire IndicatorRepository
  - Create worldWideAverage(String code) to compute average for 2017 for code
- Create IndicatorController
  - Autowire IndicatorService
  - Add @RequestMapping @RestController
  - Create avg to return the average from IndicatorService.worldWideAverage()
- Add aboveMin and aboveAverage messages to IndicatorService.
- Add above and aboveAvg mappings to IndicatorController

Automated Testing

JUnit

- Advantages of Unit Testing
  - No Runtime to start
  - Repeatable
  - IoC makes it easy to isolate and test business operations etc.
- Add the testing starter to the test dependencies

```
testCompile('org.springframework.boot:spring-boot-starter-test')
```

- Identify classes that can be effectively tested with JUnit
- Create a class in the src/test/java/*matching.package.name*/*ClassName*Test
- Annotate test methods with @Test
- Use @Before annotation to initialize test date
- Customer assert* or Hamcrest to verify conditions in each test.

Demo/ Lab 7 Junit

1. Use Intellij to automatically create JUnit for Customer class
2. Create a Before condition to set up a cal and Customer field for test data
3. Fill in each of the getters and setters with asserts etc.

Spring Testing - Services

@RunWith(SpringRunner.class) - Loads the Spring ApplicationContext

@TestConfiguration - Defines an implementation of an interface under test that can will be used by the test.

@MockBean - Bean that will be defined by Mockito to provide dependant data

Mockito - Fluent API for providing data to the service

https://docs.spring.io/spring-boot/docs/current/reference/html/boot-features-testing.html

Demo 8 : Spring Test Services

- Mock the Spring Framework and DI `@RunWith(SpringRunner.class)`
- Setup Test Configuration for Service so that it is found by the Container
- Autowire the service to the Unit Test
- Mock that dependencies (JPA or NOSQL Repositories) `@MockBean`
- Setup the test Mock repository data in the @Before
- Write the test for findInZipcode()

Lab 8: Test Service

- Mock the Spring Framework and DI `@RunWith(SpringRunner.class)`
- Setup Test Configuration for Service so that it is found by the Container
- Autowire the service to the Unit Test
- Mock that dependencies (JPA or NOSQL Repositories) `@MockBean`
- Setup the test Mock repository data in the @Before
- Write the test for findInZipcode()
- Write unit tests for the other methods

Spring Testing - REST

- Slice Testing: Comfortable space between the complexity of full Integration testing and simplicity of Unit testing
- Spring Boot Slices: REST/MVC, JPA, JDBC, etc.
- Provides SpringApplicationContext

REST Testing

- Useful for testing HTTP REST interface while mocking the data.
- Don’t have to worry about setting up the database or web service that can all be mocked
- Create a test class that sets up the data using your services and repositories
- Call MockMVC and to send URL Paths, JSON content, and query parameters and test the results.

Demo 9: REST Testing Part 1

1. Create a CustomerControllerTest Class in the test classpath in the same package as the real CustomerController.
2. Annotate with test annotations @RunWith, @SpringBootTest, @WebApplicationContext
3. Create a JSON_CONTENT_TYPE MediaType to be used later
4. Autowire the WebApplicationContext
5. Wire up the CustomerRepository to prep for tests.
6. Add test data using the repository in the setup() method
7. Create a setup (annotated with @Before) and initialize the mockMvc
8. Create tests for some REST entry points using mockMvc

Lab 9: Test Your REST

1. Create CustomerControllerTest with proper annotations to Mock the SpringBoot Context etc.
2. Setup Static types for validation (i.e. CONTENT_TYPE)
3. Autowire the webApplicaitonContext
4. Autowire the CustomerRepository
5. Autowire setConverters (see example)
6. Create a @Before method that creates two Customers and saves them to the repository
7. Store the Customer objects for test validation
8. Create tests for each of the REST Entry points
9. Write validation for each test.
10. Run test target from gradle

Nested Classes

Nested Objects

- JSON and Cassandra support nested classes
- When using JPA associations use care to avoid recursive JSON
- Cassandra - @UserDefinedType
- Useful for Cassandra list, set or map
- Need to enable custom mapping contexts in the CassandraConfig

Demo 10: Nested Objects

- Create a new Request class in the model package that includes fields for countryCode and indexCode
- Add a field to Customer.java for to map Date to requests called history.
- Add a method to log the requests by a user based on email to the CustomerService.java interface and CustomerServiceImpl.java class
- Update the CassandraConfig.java to include a mappingContext()
- Drop customer table and re-run service to recreate

```
cqlsh> describe customer;
CREATE TABLE msci.customer (
    id uuid PRIMARY KEY,
    birthday timestamp,
    email text,
    firstname text,
    history map<timestamp, frozen<request>>,
    lastname text,
    zipcode int
)
```

Lab 10: Nested Objects

- Create a new Request class in the model package that includes fields for countryCode and indexCode
- Add a field to Customer.java for to map Date to requests called history.
- Add a method to log the requests by a user based on email to the CustomerService.java interface and CustomerServiceImpl.java class
- Update the CassandraConfig.java to include a mappingContext()
- Drop customer table and re-run service to recreate
- Add findByCountryCodeAndIndicatorCode to IndicatorRepository
- Add getIndicator method to IndicatorService and IndicatorServiceImpl that takes a countryCode, indexCode, and email as parameters and returns the Indicator after logging the request using the CustomerService.
- Add a @GetMapping method that gets the countryCode/indexCode/ with the customer email as RequestParam and calls the IndicatorService.getIndicator and returns the resulting indicator
- Re-add customer and test `GET http://localhost:8080/indicators/BGR/IP.PAT.RESD/idx?email=ts@example.com`

Custom JSON

Demo 10 Date Serialization Issues

```
@Test
public void testGetCustomer() throws Exception {
  mockMvc.perform(get("/customers/" + customer1.getId()).contentType(JSON_CONTENT_TYPE))
        .andExpect(status().isOk())
        .andExpect(content().contentType(JSON_CONTENT_TYPE))
        .andExpect(content().json(json(customer1)));
}
```

Custom JSON Serialization

- Customize Jackson's serialization and deserialization to be compatible with other systems.
- Date and other more complex data structures.
- Custom Serialization can be annotated on the model
- Some custom serialization can be configured in application.properties (i.e. spring.jackson.date-format)
- @JsonFormat - specify custom format for a Date field. Shape and pattern parameters.
- @JsonDeserialize - use custom JsonDeseralizer
- @JsonSerialize - customer custom JsonSerializer

Demo 11: Change Date format in birthdayField

1. Add @JsonFormat to Customer.birthday
2. Retest with testGetCustomer

```
@Column
@JsonFormat(pattern = "MM-dd-yyyy")
private Date birthday;
```

Full Custom JSON Serialization

- Custom Serializer - Extend JsonSerializer<>
    - Override serialize
    - Use JsonGenerator parameter to generate text for value passed in
- Custom Deserializer - Extend JsonDeserializer<>
    - Override deserialize
    - Customer the JsonParser to generate an object to return
- Use with the ObjectMapper or use @JsonSerialize and @JsonDeserialize annotations

Demo 12: Custom Serializer

1. Create Color Class with red, green, and blue fields in the model package
2. Add color field to Gadget with @ManyToOne and @JoinColumn references
3. Create ColorSeralizer and ColorDeseralizer classes
4. Add `@JsonSerialize(using = ColorSeralizer.class)` and `@JsonDeserialize(using = ColorDeseralizer.class)` annotations to Color

Demo 12: Code

```
@JsonSerialize(using = ColorSeralizer.class)
@JsonDeserialize(using = ColorDeseralizer.class)
public class Color {
  @Id
  @GeneratedValue
  private Long id;

  private int red;
  private int green;
  private int blue;
```

```
public class ColorDeseralizer extends JsonDeserializer<Color> {

  @Override
  public Color deserialize(JsonParser p, DeserializationContext ctxt) throws IOException, JsonProcessingException {
     JsonNode node = p.getCodec().readTree(p);
     String rgb = node.get("rgb").textValue();
     Color color = new Color();
     color.setRed(Integer.valueOf(rgb.substring(0,2), 16));
     color.setBlue(Integer.valueOf(rgb.substring(2,4), 16));
     color.setGreen(Integer.valueOf(rgb.substring(4,6), 16));
     return color;
  }
}
```

```
public class ColorSeralizer extends JsonSerializer<Color> {
  @Override
  public void serialize(Color value, JsonGenerator gen, SerializerProvider provider) throws IOException {
     gen.writeStartObject();
     gen.writeStringField("rgb", String.format("%02X%02X%02X", value.getRed(), value.getGreen(), value.getBlue()));
     gen.writeEndObject();
  }
}
```

Lab 11-12: Custom Serializer

1. Create RequestSerializer.java to convert the Request fields as a colon seperate string *“CountryCode:IndexCode”*
2. Create RequestDeseralizer.java to split the colon separated string into the Request object fields
3. Add `@JsonSerialize(using = RequestSeralizer.class)` and `@JsonDeserialize(using = RequestDeseralizer.class)` annotations to Request

Validation and Exception Handling

Exception Handling

- Standard Exceptions are JSON but very difficult to handle in a REST app
- Provide Error Responses that match your specific API
- Be Consistent to users don’t have to handle multiple error messages.
- Use ControllerAdvice in Spring Boot 3.2 +
- The @ControllerAdvice annotation allows you to standardize Exception Handling throughout the entire app
- Use the @ExceptionHandler annotation to specify ways to handle each Exception type
- Use Custom Exceptions to give Application Specific Error Responses

Demo 13 Exception Handling

1. Demo standard exception response to invalid user ID.
2. Create a NoSuchElementResponse Class to return as the JSON result of an NoSuchElementExcepton
3. Create a ExceptionAdvice with @ControllerAdvice annotation
4. Create a noSuchElement Method with @ExceptionHanlder annotation
5. Create a ResourceNotFoundResponse class with reason, id, and className fields
6. Create a ResourceNotFoundException class with a ResourceNotFoundResponse Field and constructor that takes the values for the Response
7. Create a resourceNotFound method with an @Exceptionhandler annotation
8. Make CustomerService.getCustomer throw ResourceNotFoundException up the chain.
9. Show more useful message in the response

Demo 13 Code

```
public class ResourceErrorResponse {
  private final String reason;
  private String className;
  private Long id;

  public ResourceErrorResponse(Long id, String name, String reason)
{
     this.id = id;
     this.className = name;
     this.reason = reason;
  }

  public String getReason() {
     return reason;
  }

  public String getClassName() {
     return className;
  }

  public Long getId() {
     return id;
  }

  @Override
  public String toString() {
     return "Error: " + reason + " on id: " + id + " for " + className;
  }
}
```

```
public class ResourceException extends Exception {
  private ResourceErrorResponse response;

  public ResourceException(Class<?> aClass, Long id) {
     super("Unable to find " + id + " for " + aClass.getName());
     response = new ResourceErrorResponse(id, aClass.getName(), "Not Found");
  }

  public ResourceErrorResponse getResponse() {
     return response;
  }
}
```

```
@ControllerAdvice
public class ExceptionAdvice {

  @ExceptionHandler(NoSuchElementException.class)
  public ResponseEntity<NoSuchElementResponse>
noSuchElement(NoSuchElementException e) {
     NoSuchElementResponse notFound = new
NoSuchElementResponse(e.getLocalizedMessage());
     return new ResponseEntity<>(notFound, HttpStatus.NOT_FOUND);
  }

  @ExceptionHandler(ResourceException.class)
  public ResponseEntity<ResourceErrorResponse>
resourceNotFound(ResourceException e) {
     ResourceErrorResponse response = e.getResponse();
     return new ResponseEntity<>(response, HttpStatus.NOT_FOUND);
  }
}
```

Lab 13

1. Create a NoSuchElementResponse Class to return as the JSON result of an NoSuchElementExcepton
2. Create a ExceptionAdvice with @ControllerAdvice annotation
3. Create a noSuchElement Method with @ExceptionHanlder annotation
4. Create a ResourceNotFoundResponse class with reason, id, and className fields
5. Create a ResourceNotFoundException class with a ResourceNotFoundResponse Field and constructor that takes the values for the Response
6. Create a resourceNotFound method with an @Exceptionhandler annotation
7. Make CustomerService.getCustomer throw ResourceNotFoundException up the chain.
8. Throw ResourceException with appropriate type and reason on CustomerService and GadgetController Methods.
9. Advanced: Create Subclasses of ResourceException and ResourceErrorResponse to handle other types of errors

AOP with Custom Annotations

Spring Custom Annotations

- AOP - Aspect Oriented Programming
  - Join Point - method call or exception handling
  - Pointcut - Way to find or filter on a Join Point (i.e. method name)
  - Advice - Action taken by the aspect at a Join Point (i.e. log something before the method is called)
- Building AOP Annotations
  - Create Aspect class to define the behavior the annotation should create
  - Create Pointcut Advice to do something at particular Join Points
  - Create Custom Annotation definition using @Target, @Retention, and @interface

Demo 14: Custom Annotations

1. Annotation to Profile Method Call Times
2. Add spring-boot-starter-aop dependency to build.gradle
3. Create Profile annotation with @Target and @Retention
4. Create ProfileAspect
    a. @Aspect - tell the system that this is an AOP Aspect definition
    b. @Component - Flag this so it’s found by the Classpath Seracher
    c. Create a Map to hold the statistics for each method
    d. Create a profileExecution() method with the @Around annotation to indicate method should be called “around” each method market with the annotation @Profile
    e. Call System.currentTimeMillis before and after joinPoint.proceed()
    f. Return the return value of proceed()
    g. Store the time in the methodStats Map and print the results
5. Add the @Profile annotation to several methods and run tests to see results

Lab 14: Custom Annotations

1. Annotation to Profile Method Call Times
2. Add spring-boot-starter-aop dependency to build.gradle
3. Create Profile annotation with @Target and @Retention
4. Create ProfileAspect
   a. @Aspect - tell the system that this is an AOP Aspect definition
   b. @Component - Flag this so it's found by the Classpath Seracher
   c. Create a Map to hold the statistics for each method
   d. Create a profileExecution() method with the @Around annotation to indicate method should be called "around" each method market with the annotation @Profile
   e. Call System.currentTimeMillis before and after joinPoint.proceed()
   f. Return the return value of proceed()
   g. Store the time in the methodStats Map and print the results
5. Add the @Profile annotation to several methods and run tests to see results

Advanced

Create a new annotation @WatchDog that will log if a annotated call takes more than 100ms to complete

Demo 14 Code

```java
@Target(ElementType.METHOD)
@Retention(RetentionPolicy.RUNTIME)
public @interface Profile {
}
```

```java
@Aspect
@Component
public class ProfileAspect {

  private Map<String, LongSummaryStatistics> methodStats = new
HashMap<>();

  @Around("@annotation(Profile)")
  public Object profileExecution(ProceedingJoinPoint joinPoint) throws
Throwable {
      long begin = System.currentTimeMillis();
      Object retVal = joinPoint.proceed();
      long end = System.currentTimeMillis();
      LongSummaryStatistics stat = methodStats.computeIfAbsent(
         joinPoint.getSignature().getName(), s -> new LongSummaryStatistics());
      stat.accept(end - begin);
      System.out.printf("\n%s: c:%d avg:%f max:%d min:%d\n",
              joinPoint.getSignature(), stat.getCount(), stat.getAverage(),
             stat.getMax(), stat.getMin());
      return retVal;
  }
}
```

Security

LDAP Integration

- WebSecurityConfigurerAdapter
  - Configure with AuthenticationmangerBuilder
    - userSerachBase
    - userSerachFilter
    - groupSerachBase
    - groupSearchFilter
    - Root
    - Ldif
- HttpSecurity
  - Http
    - authorizeRequests
    - httpBasic
    - authenticationEntryPoint
- BasicAuthenticationEntryPoint

Demo 15: LDAP Integration

- LDIF
- Create SecuirtyConfig.java
- Create AuthenticationEntryPoint
- Keeping passwords private
    - rest-client.private.env.json
    - Indicators.http {username} {password}

Lab 15: LDAP Integration

- Create LDIF
- Create SecuirtyConfig.java
- Create AuthenticationEntryPoint
- Create rest-client.private.env.json
- Indicators.http {username} {password} or use browser
- Verify you cannot access without basic Authorization
- Verify you cannot access with invalid username or password
- Verify you can access as tsmith and correct password
- Create an admin group
  - Add to LDIF
  - Protect /customers so that only admins can see
  - Create additional user not in admin to test
  - Verify only admin can see /customers

Authorization

- Principal parameter on RequestMapping method
- @AuthenticationPrincipal parameter with UserDetails subclass
- SecurityContext getAuthentication()
- Authentication.getPrincipal()

Demo 16: whoami

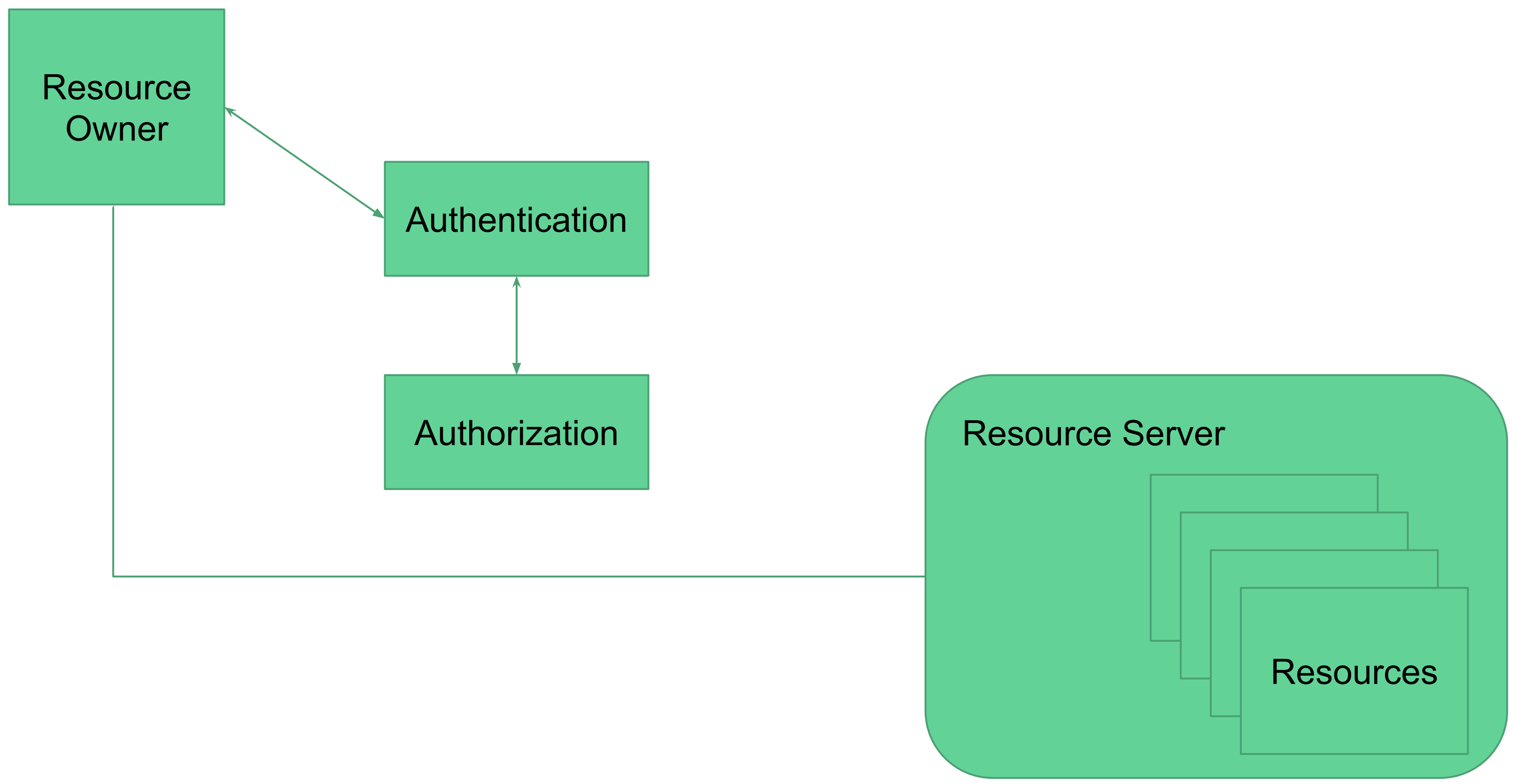
- Create RequestMapping for whoami in CustomerController
- Add Princiapal and UserDetails parameters
- UseSecurityContext
- Create string with username from the Principal and UserDetails objects

Lab 16: Authorization

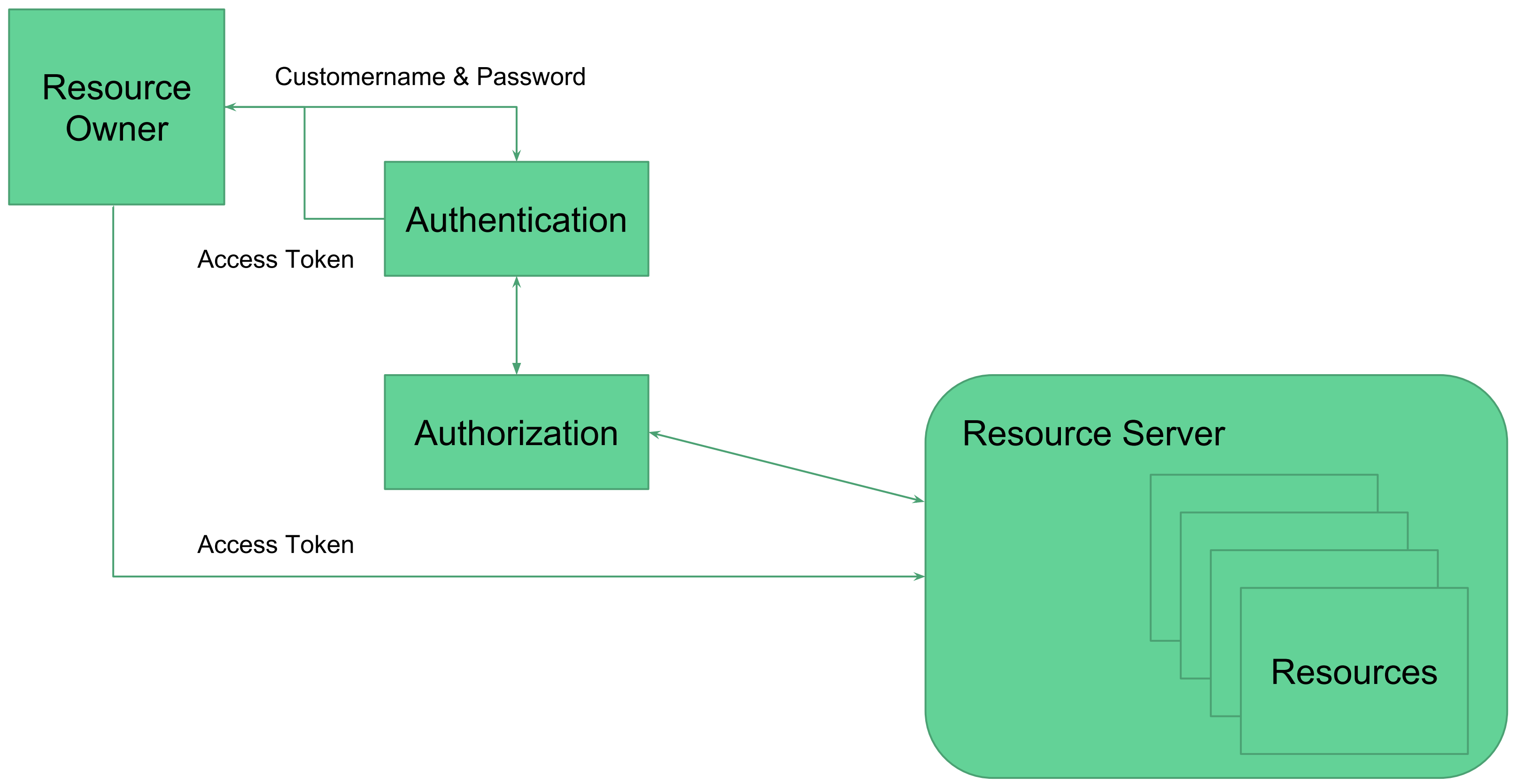
Use Principal username instead of email to track requests.

1. Add username to Customer class
2. Add findByUsername to CustomerRepostiory
3. Add logRequestByUsername to CustomerService and CustomerServiceImpl
4. Change IndicatorService.getIndicator to take a username instead of email
5. Change IndicatorController, removing email, to use Principal.
6. Add username and Authorization to customer .http file
7. Drop customers database in CQL
8. Re-add using addCustomers.http file

Spring Boot Security with OAuth2



Spring Boot Security with OAuth



User Authentication

- CustomerDetails Interface - Defines username, password, enabled etc.
- CustomerDetailsService @Service(“userDetailsService”) - provides ability to load a user by username so that password and credentials can be checked.
- AuthorizationServerConfigurerAdapter base class @EnableAuthorizationServer
  - define passwordEncoder
  - Hook up passwordEncoder
  - Configure authentication manager and userDetailsService
  - Configure clients
- Can be outside resource (i.e. Facebook, Google, GitHub, etc)
- Users Passes Credentials and Receives and receives an Access Token, Refresh Token, Expiration, and scope.

Authorization

- Restrict which Authenticated users can do what with the resources
- Defined in a Resource Server configuration class. @EnableResourceServer that extends ResourceServerConfigurerAdapter
- Configure HttpSecurity - Create rules for requests that match URLs
- Configure Resource IDs
- Implement AuthorizationServerConfigurerAdapter - Define clients, secret codes, expiration, scope, grant types, and resourceIds

Demo 17: Spring Security with OAuth2

1. Add `compile('org.springframework.boot:org.springframework.security.oauth')` and `compile('org.springframework.boot:spring-boot-starter-security')` dependencies to build.gradle
2. Create a OAuth2Config that extends AuthorizationServiceConfigurerAdapter
3. Create CustomerSecurity that implements CustomerDetailsService interface
4. Add findOneByUsername to CustomerRepository
5. Make Customer implement UserDetails, add new columns for security
6. Add ResourceServerConfig that extends ResourceServerConfigurerAdapter
7. Add WebSecurityConfig that extends WebSecurityConfigurerAdapter
8. Add @EnableResourceServer to GadgetsApplication
9. Create SQL resources with users and passwords, schema.sql and data.sql

Demo 17: Rest JSON Output

```
POST
http://localhost:8080/oauth/token?grant_type=password&username=geoff@example.com&password=password
Accept: application/json
Authorization: Basic Y29ycDpzZWNyZXQ=
```

```
{
  "access_token": "1f3f68ae-78dc-4fdd-bcf7-f48af3fa1fd7",
  "token_type": "bearer",
  "refresh_token": "3e1efefa-1b45-4261-8738-a6af919e0462",
  "expires_in": 3482,
  "scope": "read write"
}
```

Demo 17: Authentication Code

```
public class Customer implements UserDetails {
   @Id
   @GeneratedValue(strategy = GenerationType.IDENTITY)
   @Column(nullable = false, updatable = false)
   private Long id;
```

```
@Service("userDetailsService")
public class CustomerSecurityService implements CustomerDetailsService {

   @Autowired
   private CustomerRepository userRepository;

   @Override
   public CustomerDetails loadCustomerByCustomername(String username) throws
CustomernameNotFoundException {
       return userRepository.findOneByUsername(username);
   }
}
```

```
@Configuration
@EnableWebSecurity
@EnableGlobalMethodSecurity(prePostEnabled = true)
public class WebSecurityConfig extends WebSecurityConfigurerAdapter {

  /**
   * Constructor disables the default security settings
   */
  public WebSecurityConfig() {
     super(true);
  }


  @Bean
  @Override
  public AuthenticationManager authenticationManagerBean() throws Exception {
     return super.authenticationManagerBean();
  }

}
```

Demo 17: Authorization Code 1 of 2

```
@Configuration
@EnableResourceServer
public class ResourceServerConfig extends ResourceServerConfigurerAdapter {

  @Override
  public void configure(ResourceServerSecurityConfigurer resources) throws Exception {
     resources.resourceId("resource");
  }

  @Override
  public void configure(HttpSecurity http) throws Exception {
     http.requestMatchers().antMatchers("/users/**")
           .and()
           .authorizeRequests()
           .anyRequest().authenticated();
  }
}
```

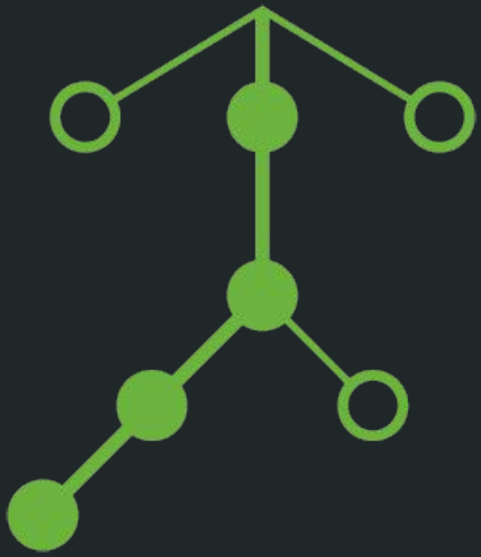
Demo 17: Authorization Code 2 of 2

```
@Configuration
@EnableAuthorizationServer
public class OAuth2Config extends AuthorizationServerConfigurerAdapter {

  // secret = secret
  private static final String CORP_SECRET_BCRYPT =
"$2a$04$DQjbLE9xtfkN3T1cq3QL.u3OKhSrstz7wbywx9kyzraOwKJXM8Y9e";

  @Autowired
  @Qualifier("userDetailsService")
  private CustomerDetailsService userDetailsService;

  @Autowired
  private AuthenticationManager authenticationManager;

  @Value("${corp.oauth.tokentTimeout:3600}")
  private int expiration;

  @Bean
  public PasswordEncoder passwordEncoder() {
     return new BCryptPasswordEncoder();
  }

  @Override
  public void configure(AuthorizationServerEndpointsConfigurer configurer) {
     configurer.authenticationManager(authenticationManager);
     configurer.userDetailsService(userDetailsService);
  }

  @Override
  public void configure(ClientDetailsServiceConfigurer clients) throws Exception {
     clients.inMemory()
           .withClient("corp")
           .secret(CORP_SECRET_BCRYPT)
           .accessTokenValiditySeconds(expiration)
           .scopes("read", "write")
           .authorizedGrantTypes("password", "refresh_token")
           .resourceIds("resource");
  }
}
```

Lab 17: Spring Security using OAuth

1. Add compile(**'org.springframework.boot:org.springframework.security.oauth'**) and compile(**'org.springframework.boot:spring-boot-starter-security'**) dependencies to build.gradle
2. Create a OAuth2Config that extends AuthorizationServiceConfigurerAdapter
3. Create CustomerSecurity that implements CustomerDetailsService interface
4. Add findOneByCustomername to CustomerRepository
5. Make Customer implement UserDetails, add new columns for security
6. Add ResourceServerConfig that extends ResourceServerConfigurerAdapter
7. Add WebSecurityConfig that extends WebSecurityConfigurerAdapter
8. Add @EnableResourceServer to GadgetsApplication
9. Create SQL resources with users and passwords, schema.sql and data.sql
10. Protect the /gadgets resources
11. Allow the corp client to access gadgets
12. Create an additional client and allow access to just the /gadgets resources

API Versioning



API Versioning

- When the “contract” needs to change
- Not always necessary if **adding** new fields to returned JSON, but some sensitive clients will break
- May be necessary if the *semantics* change even if the *syntax* does not
- Four popular approaches
    - URI Versioning
    - Request Parameter Versioning
    - Custom Header Versioning
    - Media Type Versioning

URI Versioning options Part 1

- URI Versioning
  - http://example.com/v1/user/123
  - http://example.com/v2/user/123
  - Best when planned ahead and can start with first version
  - Twitter -> https://api.twitter.com/1.1/search/tweets.json
- Request Param Versioning
  - http://example.com/user/123?version=1
  - http://example.com/user/123?version=2
  - Messy and easily forgotten
  - Amazon -> https://sdb.amazonaws.com/?Action=PutAttributes...&Version=2009-04-15...
  - Can get long "polluting the URL" but good to add if you deploy V1 before you think about versioning

API Versioning Options Part 2

- Headers
    - Client is required to put the desired version in the header
    - Check using `@RequestMapping(headers = "X-API-VERSION=2")`
    - Cannot test/explore using regular web browser
    - Microsoft does this
- Media Type
    - Client puts in the “Accepts” header
    - Accept=application/vnd.company.app-v1+json
    - Check using `@RequestMapping(produces = "application/vnd.company.app-v1+json")`
    - Cannot test/explore using regular web browser
    - GetHub does this

Lab 13: API Versioning

- Create a new version of add /users/older that returns just a list of user IDs instead of full objects
- Use the version technique that best fits the use case for your users.

Wrap Up

Wrap up, Final Q&A

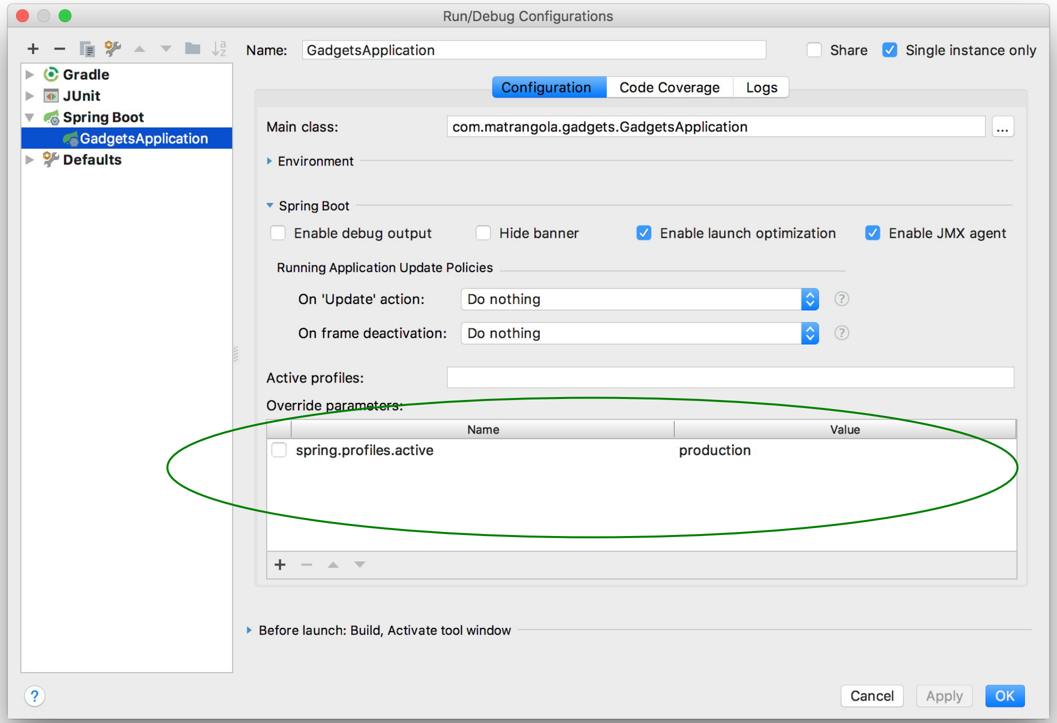
- Review final version of the Gadget App
- Remaining Questions

Supplemental: Dynamic Configurations and Logging

Dynamic Configurations and Logging

```
spring:
 jpa:
   hibernate:
     ddl-auto: update
   show-sql: true
 datasource:
   url: jdbc:mysql://localhost:3306/gadget
   username: db
   password: spring
---
spring:
 profiles: development
logging:
 level:
   org.hibernate.SQL: DEBUG
   org.hibernate.type.descriptor.sql.BasicBinder: TRACE
---
spring:
 profiles: production
logging:
 level:
   org.hibernate.SQL: WARN
 file: logs/log.txt
```

IDE & Command Line



Run/Debug Configurations

Name: GadgetsApplication

Share [ ] Single instance only [✓]

Main class: com.matrangola.gadgets.GadgetsApplication

Environment

Spring Boot

Enable debug output [ ] Hide banner [ ] Enable launch optimization [✓] Enable JMX agent [✓]

Running Application Update Policies

On 'Update' action: Do nothing

On frame deactivation: Do nothing

Active profiles:

Override parameters:

Name	Value
spring.profiles.active	production

Before launch: Build, Activate tool window

```
java -jar -Dspring.profiles.active=production demo-0.0.1-SNAPSHOT.jar
```

SLF4J

- Facade for logging systems
- Allows different Logging based on “user” configuration

```
implementation 'org.slf4j:slf4j-api:1.7.25'
```

```
Logger LOG = LoggerFactory.getLogger(CustomerController.class);
```

```
@Profile
@CrossOrigin(origins = "http://localhost:9000")
@RequestMapping("/")
public List<Customer> get() {
   LOG.trace("get");
   return userService.getCustomers();
}
```

Supplemental Interceptors

Spring Interceptors

- Extend HandlerInterceptorAdaptor
- Override preHandle postHandle etc

Configuration

- `@Configuration public class WebMvcConfig extends WebMvcConfigurerAdapter`
- `@Autowired HandlerInterceptor yourInjectedInterceptor;`
- `@Override public void addInterceptors(InterceptorRegistry registry) {`