Fast Track to Spring Boot

Intermediate Spring Boot REST Services - 4 Day

Overview

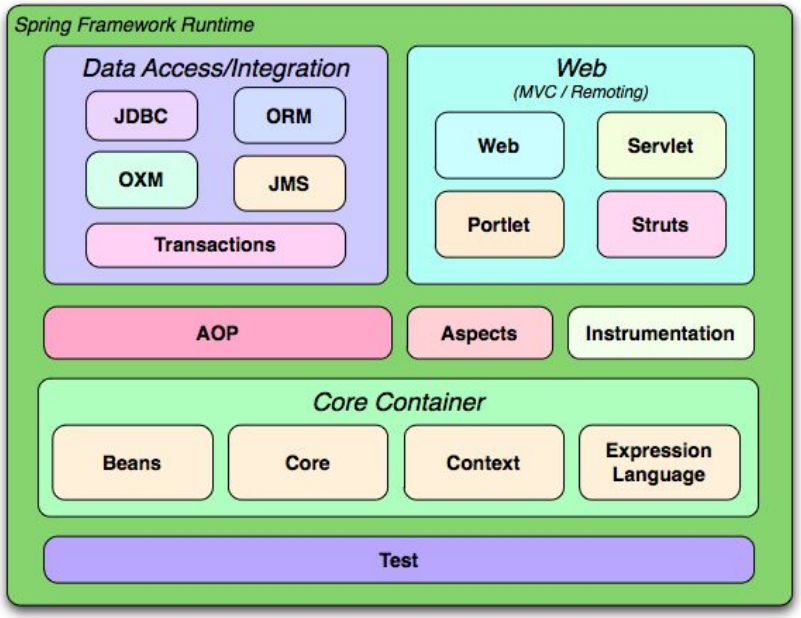
- Introductions
  - Instructor - Geoff Matrangola - geoff@matrangola.com @triglm
  - Company - DevelopIntelligance http://www.developintelligence.com/ (show 2 slides)
  - Students - Names, Current projects, Class Expectations
  - Course - How to develop a Rest API Using Spring
- Logistics
  - Start, end, break times
  - Facilities
- Class Agenda
  - See Class outline
- Class Flow
  - Slides
  - Demo
  - Lab

What is 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 Tomcat
- Easy Database configuration
- Spring - Dependency Injection and Inversion of Control

Key Elements of the Spring Framework

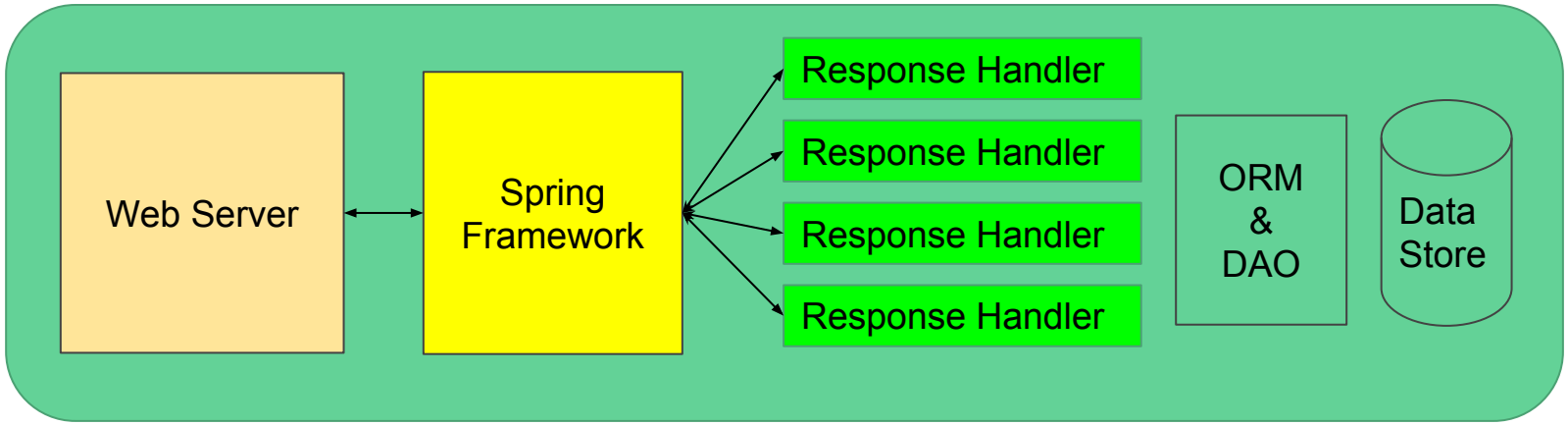
- Modules
- Core Container
- Beans
- Context
- AOP
- Other - Data, Web, Test, Instrumentation



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.

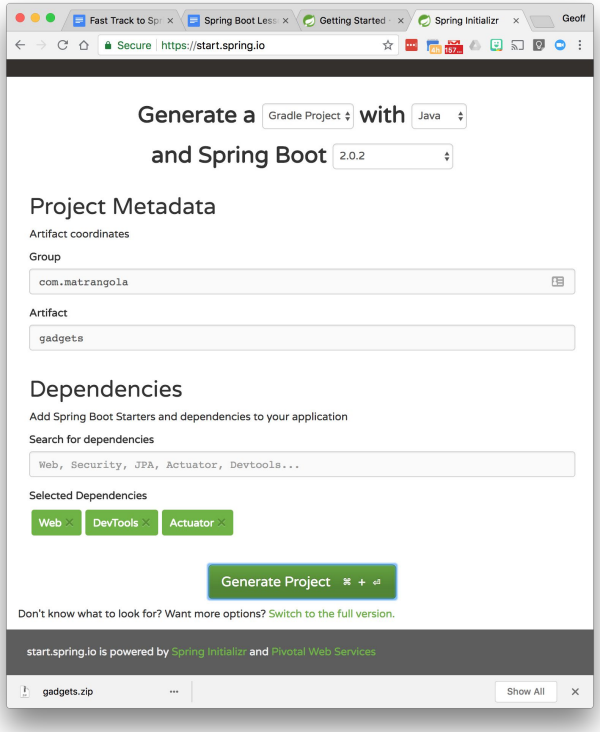
Setup

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

Spring Initializr

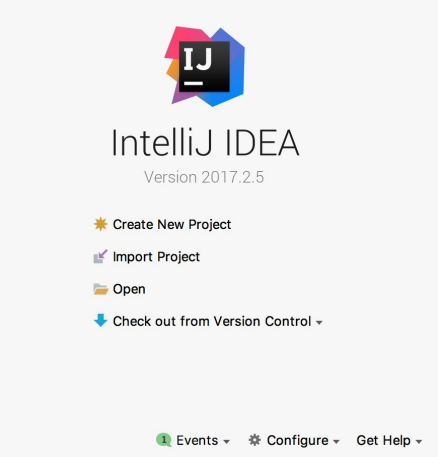
https://start.spring.io/

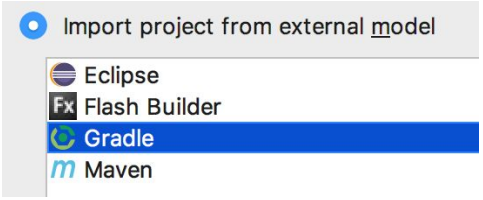
- Gradle Project
- Java
- 2.0.2
- Group: com.whatever
- Artifact: gadgets
- Dependencies: Web, Actuator, DevTools
- Download
- Unzip



Import Part 1

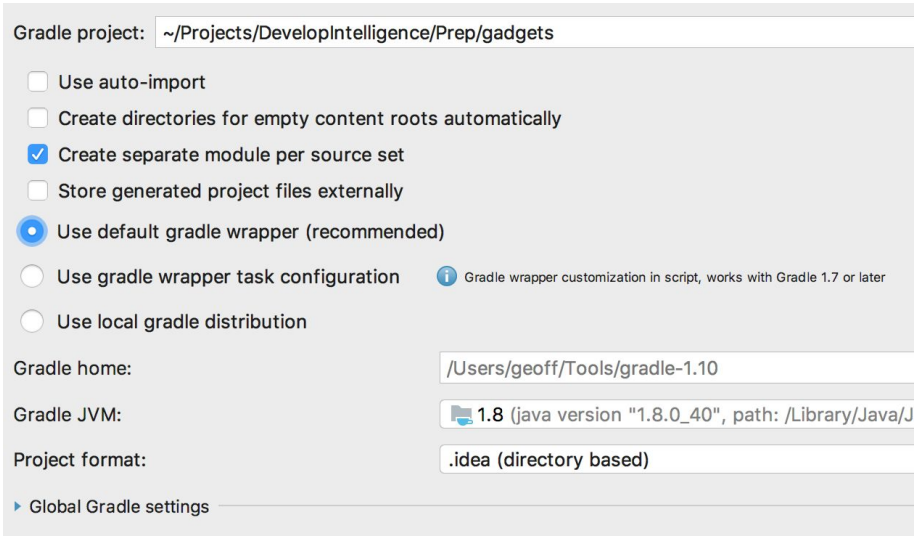
- Launch IntelliJ Idea
- Import Project
- Select Downloaded & Unzipped Directory
- Select Import project...
- Gradle





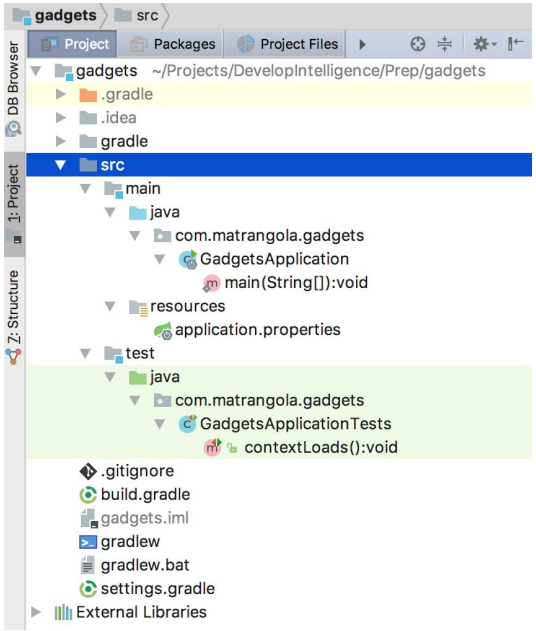
Import Part 2

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



Project Structure

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



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

Live Demo

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

public class User {
  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;
  }
}
```

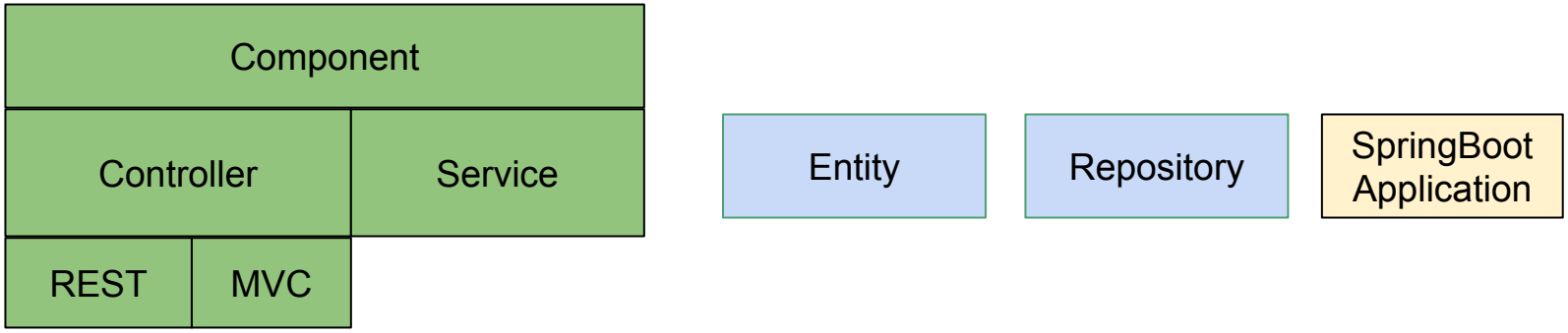
```
@RestController
public class UserController {
  @RequestMapping("/makeUser")
  public User greeting(@RequestParam(value="last") String lastName,
                       @RequestParam(value="first") String firstName) {
    User user = new User();
    user.setFirstName(firstName);
    user.setLastName(lastName);
    return user;
  }
}
```

Lab 1

1. Specify and download Spring Initilizer
2. Unzip
3. Import into IntelliJ
4. Use annotations to create ResetController
5. Test with rest runner

Core Spring Boot Components and Classes

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.

Demo 2: Service Interface & Class

- Create new service package
- Add UserService Interface
- Add UserServiceImpl
- Wire up the UserService to the UserController

Advanced:

- Add a delete field to the UserService

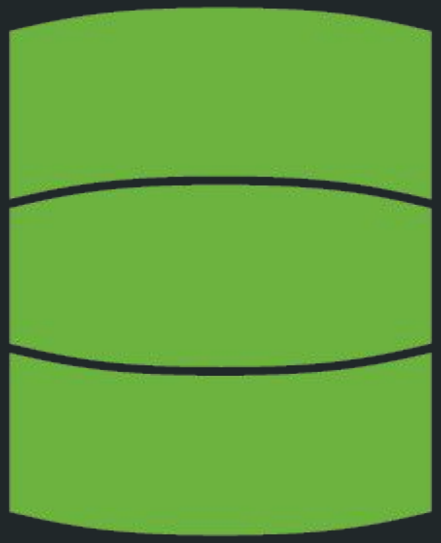
Live Demo 2

```
@Service
public class UserServiceImpl implements
UserService {
   private Set<User> users = new HashSet<>();

   @Override
   public void addUser(User user) {
      users.add(user); // todo check if exists and
throw exception if alread there, etc.
   }

   @Override
   public void updateUser(User user) {
      users.add(user); // todo check if exists and
throw exception if not, etc
   }
}
```

```
@RestController
public class UserController {
   @Autowired
   UserService userService;

   @RequestMapping("/makeUser")
   public User greeting(
         @RequestParam(value="last") String lastName,
         @RequestParam(value="first") String firstName) {
      User user = new User();
      user.setFirstName(firstName);
      user.setLastName(lastName);
      userService.addUser(user);
      return user;
   }
}
```

```
public interface UserService {
   void addUser(User user);
   void updateUser(User user);
}
```

Data Management



Data Management

- Entity - ORM Table mapping
  - Defines Primary Key
  - Fields
  - Relationships to other tables
  - Indexing
  - Data integrity rules
  - Maps to SQL Database
- DAO
  - JpaRepository
  - Interfaces that can be used by services to access Entities in the Data Store
  - SQL code automatically generated by rule-based interface

Java Persistence - Some Annotations used

@Entitiy - Mark a class as stored in the database

@Table - Define the Table where the Entity is stored

@Column - Field is used as a column

@Id - Primary Key column

@GeneratedValue - indicate that the column is automatically generated

Live Demo 3

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

```
@Service
public class UserServiceImpl implements UserService {
   @Autowired
   private UserRepository userRepository;

   @Override
   public void addUser(User user) {
      userRepository.save(user); // todo check if exists
and throw exception if alread there, etc.
   }

   @Override
   public void updateUser(User user) {
      userRepository.save(user); // todo check if exists
and throw exception if not, etc
   }

   @Override
   public void deleteUser(User user) {
      userRepository.deleteById(user.getId());
   }
}
```

```
@Entity
@Table(name = "user")
public class User {
   @Id
   @GeneratedValue
   private Long id;

   @Column
   private String firstName;
   @Column
   private String lastName;

   public Long getId() {
      return id;
   }

// ...
```

Data Management Demo Results: Need Config

```
Error starting ApplicationContext. To display the conditions report re-run your application with 'debug' enabled.
2018-05-15 20:53:18.602 ERROR 23759 --- [  restartedMain] o.s.b.d.LoggingFailureAnalysisReporter   :

***************************
APPLICATION FAILED TO START
***************************

Description:

Failed to configure a DataSource: 'url' attribute is not specified and no embedded datasource could be configured.

Reason: Failed to determine a suitable driver class


Action:

Consider the following:
        If you want an embedded database (H2, HSQL or Derby), please put it on the classpath.
        If you have database settings to be loaded from a particular profile you may need to activate it (no profiles are currently active).

Disconnected from the target VM, address: '127.0.0.1:52657', transport: 'socket'

Process finished with exit code 0
```

Spring Configuration Management

resources directory

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

Live Demo 3b: Setup MySQL

```
$ sudo mysql --password

mysql> create database gadget;
Query OK, 1 row affected (0.04 sec)

mysql> create user 'db'@'localhost' identified by 'spring';
Query OK, 0 rows affected (0.04 sec)

mysql> grant all on gadget.* to 'db'@'localhost';
Query OK, 0 rows affected (0.01 sec)
```

Live Demo: Config

```
compile('mysql:mysql-connector-java')
```

```
spring.jpa.hibernate.ddl-auto=create
spring.datasource.url=jdbc:mysql://localhost:3306/gadget
spring.datasource.username=db
spring.datasource.password=spring
```

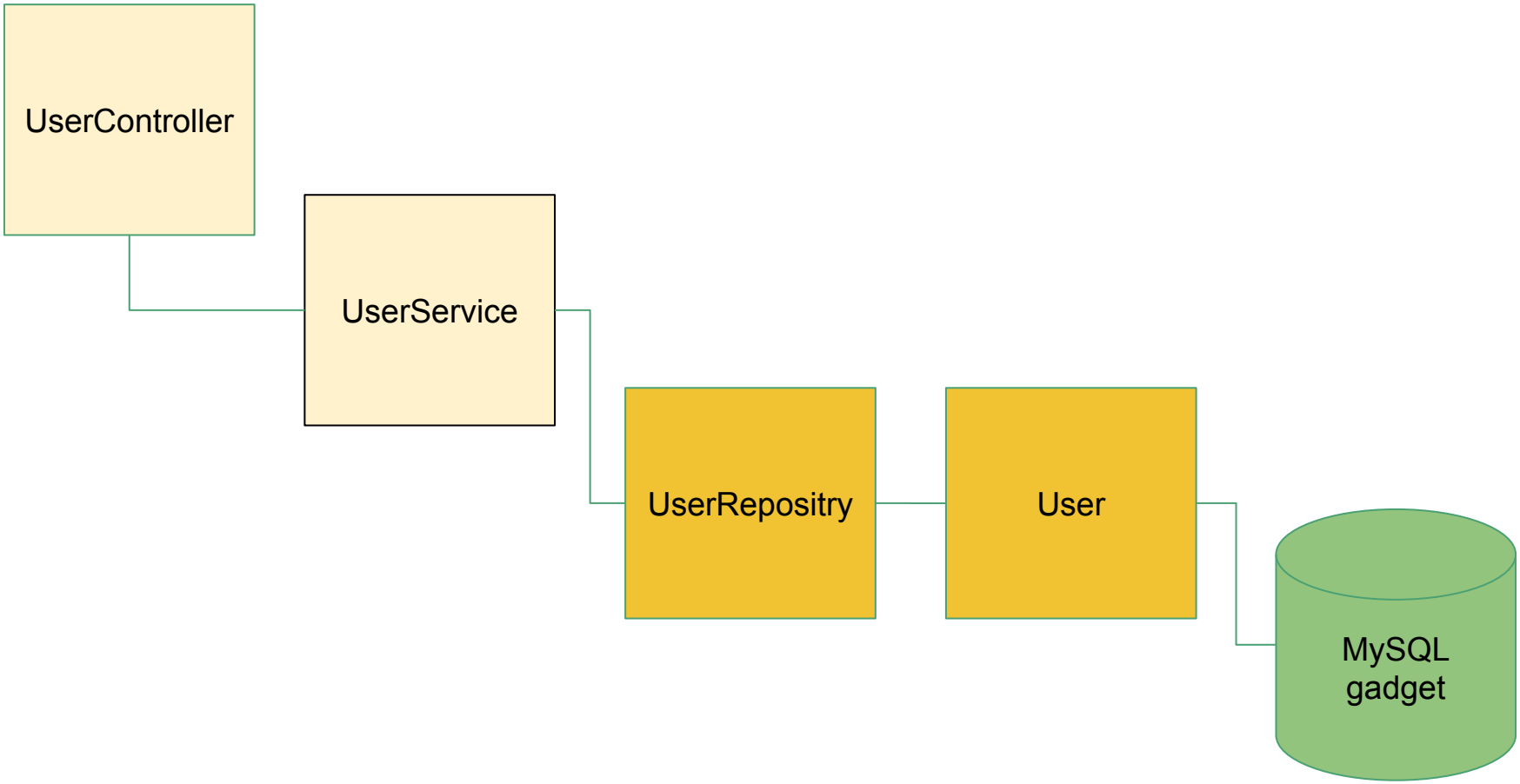
Live Demo: Select Table

```
mysql> use gadget
Reading table information for completion of table and column names
You can turn off this feature to get a quicker startup with -A

Database changed
mysql> show tables;
+--------------------+
| Tables_in_gadget   |
+--------------------+
| hibernate_sequence |
| user               |
+--------------------+
2 rows in set (0.00 sec)

mysql> select * from user;
+----+------------+------------+
| id | first_name | last_name  |
+----+------------+------------+
|  1 | Geoff      | Matrangola |
+----+------------+------------+
1 row in set (0.01 sec)
```

Lab 3: Data Management

1. Add Entity Annotations and id to User Class
2. Add UserRepository
3. Replace HashMap in UserService
4. Setup MySQL database
5. Create applicaiton.properties

Gadget App Summary



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

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 UserController {
  //...
}
```

@RequestMapping

```
@RestController
@RequestMapping("/users")
public class UserController {
        @RequestMapping("/makeUser")
        public User 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 User add(@RequestParam(value="last") String lastName,
        @RequestParam(value="first") String firstName,
        @RequestParam(name = "birthday", required = false) String birthdayText) // ...


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

@RequestBody

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

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

Demo/Lab 4: RequestMapping

Demo 4 - Request Mapping

- Modify User to add Birthday so that we can have more data to play with
- Add a getUsers that passes through the UserRepository to get a list of users to satisfy User requests.
- Add class level RequestMapping for UserController and root mapping to get a list of all users
- Add temporary test method to create bogus users to return
- Change “greeting” method name to “add”.
- Change “/makeUser” to “/new”, add RequestType.GET
- Add another “new”/add that takes a User as a @RequestBody parameter and has a RequestType.PUT
- Make and update() that has a RequestType.POST
- Make a older() RequestMapping that returns older than PathVariable
- Make a foo() with crazy RequestMapping value strings for wildcards

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 SimpleDateFormate to convert it to a java.util.Date.
3. Set the Date in the User 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 user by ID (DB PK) using a @PathVariable

Advanced:

- Add a new RequestMapping that accepts an image and returns a string expressing the size in bytes.

Lab 4: Solution Part 1

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

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

  return "User ID: " + userId + " uploaded " + bytes.length + " bytes";
}
```

Lab 4 Solution Part 2

```
@RequestMapping("/{id}")
public User getById(@PathVariable("id") long id) {
   return userService.getUser(id);
}
```

```
@Override
public User getUser(long id) {
   return userRepository.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 5: Code Review

Refactor

1. path = “/users”, consumes and produces default for class
2. value -> path
3. Add RequestMethod

Lab 5: Code Cleanup

Update your code to conform to the API standards that were shown in the demo.

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
- User assert* or Hamcrest to verify conditions in each test.

Demo/ Lab 6

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

Spring Boot 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 7: REST Testing Part 1

1. Create a UserControllerTest Class in the test classpath in the same package as the real UserController.
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 UserRepository 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 REST entry points using mockMvc

Lab 7 Test Your REST

1. Create UserControllerTest 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 UserRepository
5. Autowire setConverters (see example)
6. Create a @Before method that creates two users and saves them to the repository
7. Store the user 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

Rest and Test review & wrap-up

Demo Date Serialization Issues

```
@Test
public void testGetUser() throws Exception {
  mockMvc.perform(get("/users/" + user1.getId()).contentType(JSON_CONTENT_TYPE))
        .andExpect(status().isOk())
        .andExpect(content().contentType(JSON_CONTENT_TYPE))
        .andExpect(content().json(json(user1)));
}
```

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 - user custom JsonSerializer

Demo 8a: Change Date format in birthdayField

1. Add @JsonFormat to User.birthday
2. Retest with testGetUser

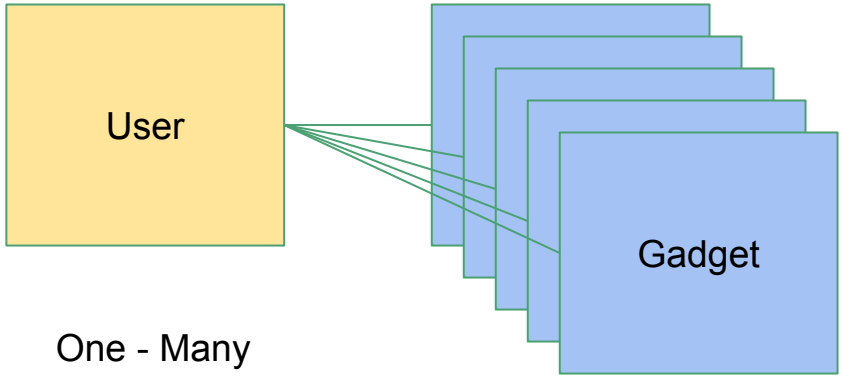
```
@Column
@JsonFormat(pattern = "MM-dd-yyyy")
private Date birthday;
```

Relational Data

- Relations are supported in JPA ORM and JSON Jackson libraries
- Many - One, Many - to - Many, One - to - One, One - to - Many
- ORM - Quick review
  - @OneToMany - Indicates a list of related objects
    - mappedBy - Field on the Other side of the association
    - Cascade - handle deletes
    - Fetch - lazy or eager
  - @ManyToOne - Indicates that there may be multiple entities referring to this object
  - @JoinColumn - name the column that is associated
- JSON - mostly automatic
  - @JsonIdentytInfo - points out the ID Class to the entity

http://www.baeldung.com/jackson-annotations

Add relational data to model



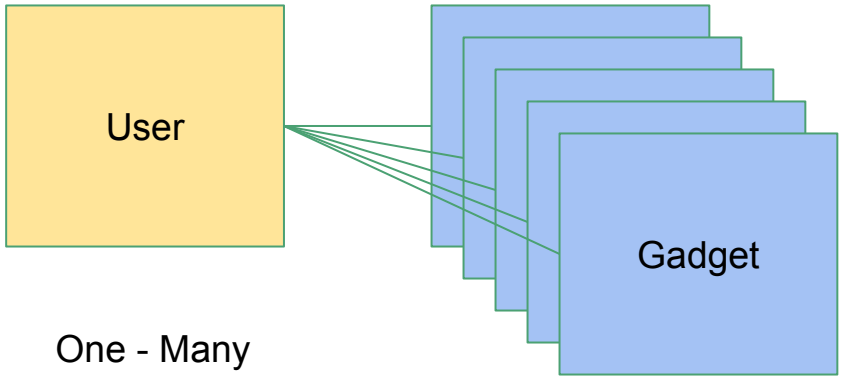

```
@Entity
@Table(name = "user")
public class User {
   // ...

   @OneToMany(mappedBy = "owner", cascade = ALL,
                        fetch = FetchType.EAGER)
   private Set<Gadget> gadgets;

// ...
```

```
@Entity
@Table(name = "gadget")
public class Gadget {
   @Id
   @GeneratedValue
   private Long id;

   @Column
   private String name;

   @Column(name = "isOn")
   private boolean on;

   @ManyToOne
   @JoinColumn(name="owner_id", nullable = false)
   private User owner;
// ….
```

Demo 8b

1. Create Gadget Model Class
2. Setup JPA Relation to User
    a. Gadget - @ManyToOne and @JoinColumn on owner
    b. User - @OneToMany on gadgets

Relational Data Recursion Problem



One - Many

```
{
   "id": 1,
   "firstName": "Geoff",
   "lastName": "Matrangola",
   "gadgets": [
    {
      "id": 2,
      "name": "Light",
      "on": false,
      "owner": {
        "id": 1,
        "firstName": "Geoff",
        "lastName": "Matrangola",
        "gadgets": [
          {
            "id": 2,
            "name": "Light",
            "on": false,
            "owner": {
              "id": 1,
              "firstName": "Geoff",
              "lastName": "Matrangola",
              "gadgets": [
                {
                  "id": 2,
                  "name": "Light",
                  "on": false,
                  "owner": {
                    "id": 1,
                    "firstName": "Geoff",
                    "lastName": "Matrangola",
//...
```

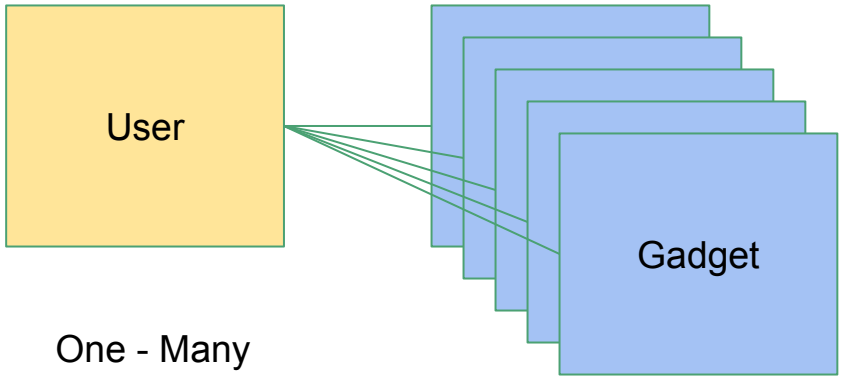
Recursive association

Demo 8c: Fix Recursion with @JsonIdentityInfo

```
@Entity
@Table(name = "gadget")
@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class,  property = "id")
public class Gadget {
   @Id
   @GeneratedValue
   private Long id;
```

```
@Entity
@Table(name = "user")
@JsonIdentityInfo(generator = ObjectIdGenerators.PropertyGenerator.class,  property = "id")
public class User {
   @Id
   @GeneratedValue
   private Long id;
```

New Output without recursion



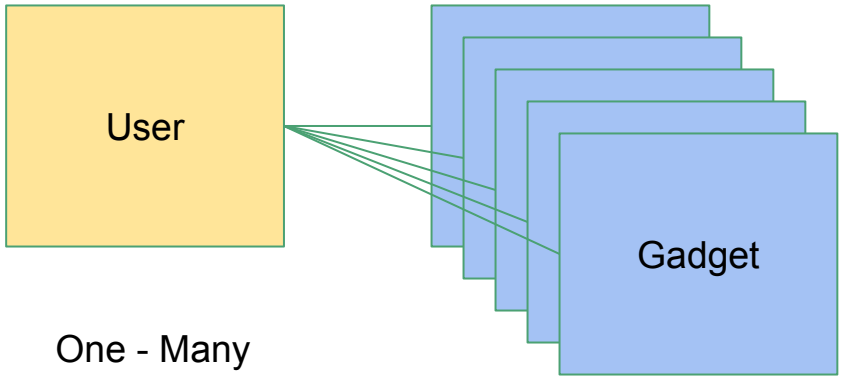

```
[
  {
    "id": 1,
    "firstName": "Geoff",
    "lastName": "Matrangola",
    "gadgets": [
      {
        "id": 2,
        "name": "Light",
        "on": false,
        "owner": 1
      }
    ],
    "birthday": null
  }
]
```

Demo 8d: Only include the IDs

```
@OneToMany(mappedBy = "owner", cascade = ALL, fetch = FetchType.EAGER)
@JsonIdentityReference(alwaysAsId = true)
private Set<Gadget> gadgets;
```

```
@ManyToOne
@JoinColumn(name="owner_id", nullable = false)
@JsonIdentityReference(alwaysAsId = true)
private User owner;
```

Final Output - Using IDs for Reference




```
[
  {
    "id": 1,
    "firstName": "Geoff",
    "lastName": "Matrangola",
    "gadgets": [
      2
    ],
    "birthday": null
  }
]
```

Lab 8: Implement Relations

1. Create Gadget Class
2. Create Id, name, on fields
3. Specify ORM Annotations @Entity, @Table, @Id, @GeneratedValue, @Column, @ManyToOne
4. Add Gadget list to User class with @OneToMany annotation
5. Specify @JsonIdentityInfo for and @JsonIdentytRefernce annotations for both classes
6. Create GadgetRepository
7. Create GadgetController with getById and add REST entry points
8. Run Http Client requests to verify implementation

Advanced:

1. Create Unit test for Gadget Class
2. Create Room Entity and supporting ORM, JSON, and REST impementation

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
  - User the JsonParser to generate an object to return
- Use with the ObjectMapper or use @JsonSerialize and @JsonDeserialize annotations

Demo 9: Custom Serializer

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

Demo 9: Code

```
@Entity
@Table(name = "color")
@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;
```

```
@ManyToOne(cascade = CascadeType.ALL)
@JoinColumn(name="color_id", nullable = true)
private Color color;
```

```
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 9: Custom Serializer

1. Create Color Class with red, green, and blue fields in the model package
2. Add ORM annotations for @Entitiy, @Table, @Id, etc
3. Add color field to Gadget with @ManyToOne and @JoinColumn references
4. Create ColorSeralizer and ColorDeseralizer classes
5. Add `@JsonSerialize(using = ColorSeralizer.class)` and `@JsonDeserialize(using = ColorDeseralizer.class)` annotations to Color
6. Add support for the id field in the serializer and deserializer

CORS - Cross Origin Requests

- CORS can be configured for entire Spring Boot App, at the Controller Level, or per method.
- For Global
  - Use @Bean and create a method that returns WebMvcConfigurer
  - Return a WebMvcConfigurerAdapter object that implements addCoresMapping
- For Fine Grained control
  - Use @CrossOrigin annotation to list the valid origins for the response for a controller class
  - Use @CrossOrigin annotation on individual RequestMapping methods.
  - origin - the URL of the origin
  - methods - if different than those specified in the RequestMapping
  - allowedHeaders - headers permitted from forwarding resource
  - exposedHeaders - headers that the client will be allowed to access on the actual response
  - allowCredentials - permit cookies and user credentials from client
  - maxAge - max age (in seconds) of the cache for responses

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 10 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 UserService.getUser throw ResourceNotFoundException up the chain.
9. Show more useful message in the response

Demo 10 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 10

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 UserService.getUser throw ResourceNotFoundException up the chain.
8. Throw ResourceException with appropriate type and reason on UserService and GadgetController Methods.
9. Advanced: Create Subclasses of ResourceException and ResourceErrorResponse to handle other types of errors

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 11: 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 11: 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

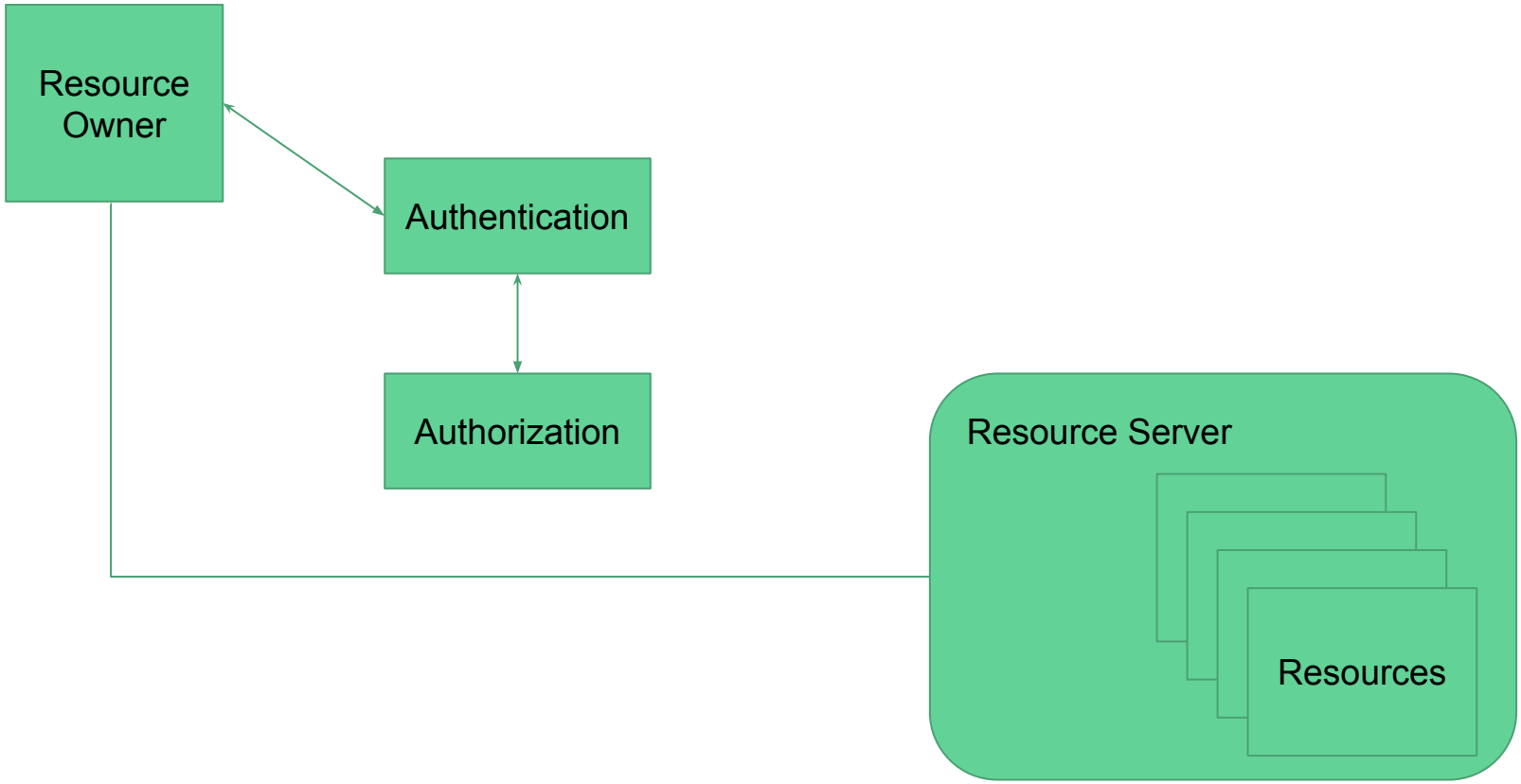
Demo 11 Code

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

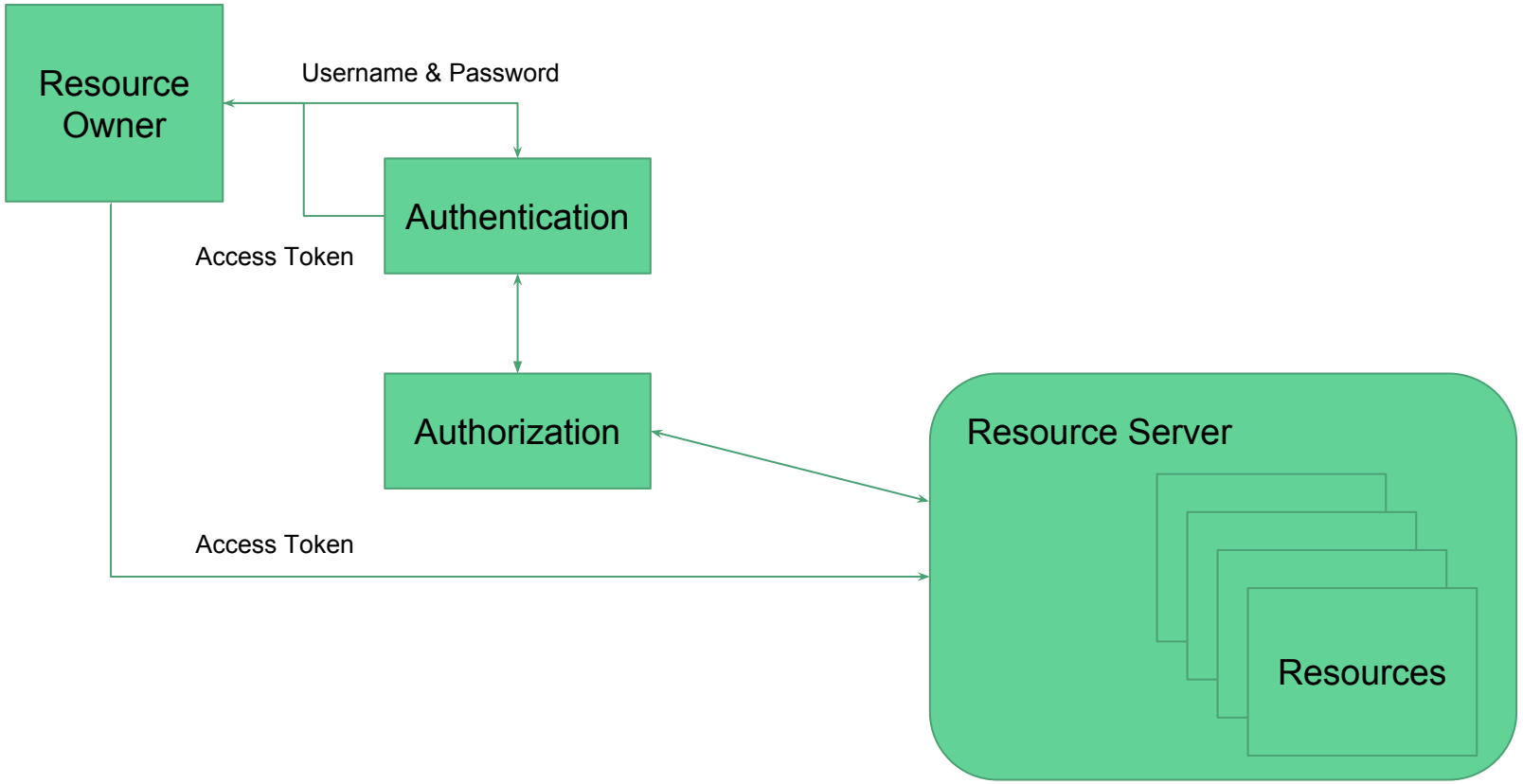
```
@Aspect
@Component
public class ProfileAspect {

   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

Spring Boot Security with OAuth2



Spring Boot Security with OAuth



User Authentication

- UserDetails Interface - Defines username, password, enabled etc.
- UserDetailsService @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 12: 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 UserSecurity that implements UserDetailsService interface
4. Add findOneByUsername to UserRepository
5. Make User 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 12: 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 12: Authentication Code

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

```
@Service("userDetailsService")
public class UserSecurityService implements UserDetailsService {

   @Autowired
   private UserRepository userRepository;

   @Override
   public UserDetails loadUserByUsername(String username) throws UsernameNotFoundException {
      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 12: 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 12: 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 UserDetailsService 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 12: 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 UserSecurity that implements UserDetailsService interface
4. Add findOneByUsername to UserRepository
5. Make User 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