Introduction to Spring

Swagger for documentation

Introduction

- Swagger is an Interface Description Language for describing RESTful APIs expressed using JSON.
- Swagger is used together with a set of open-source software tools to design, build, document, and use RESTful web services.
- Swagger includes automated documentation, code generation (into many programming languages), and test-case generation.

Purpose

1. Documenting APIs
2. Developing APIs
3. Interacting with APIs

Documenting APIs

When described by an OpenAPI document, Swagger open-source tooling may be used to interact directly with the API through the Swagger UI. This project allows connections directly to live APIs through an interactive, HTML-based user interface. Requests can be made directly from the UI and the options explored by the user of the interface.

Useful links

- Specification: https://swagger.io/specification/v2/
- https://editor.swagger.io/

Add Maven dependency

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-boot-starter</artifactId>
    <version>3.0.0</version>
</dependency>
```

```
<dependency>
    <groupId>org.springframework.boot</groupId>
    <artifactId>spring-boot-starter-parent</artifactId>
    <version>2.4.0</version>
</dependency>
```

Configure

```
@Configuration
public class SpringFoxConfig {
    @Bean
    public Docket api() {
        return new Docket(DocumentationType.SWAGGER_2)
          .select()
          .apis(RequestHandlerSelectors.any())
          .paths(PathSelectors.any())
          .build();
    }
}
```

Add UI for Swagger

```
<dependency>
    <groupId>io.springfox</groupId>
    <artifactId>springfox-swagger-ui</artifactId>
    <version>2.9.2</version>
</dependency>
```

Useful annotations

- @ApiOperation
- @Consumes
- @Produces

HTTP Connection (Java Client)

Intro

- An HttpClient can be used to send requests and retrieve their responses.
- Java contains support for an embedded http client
- Starting with JDK 9 and supports HTTP/2 with backward compatibility still facilitating HTTP/1.1.

HTTP Client example < java 9

```
HttpClient httpclient = HttpClients.createDefault();
HttpPost httppost = new HttpPost("http://www.a-domain.com/foo/");

// Request parameters and other properties.
List<NameValuePair> params = new ArrayList<NameValuePair>(2);
params.add(new BasicNameValuePair("param-1", "12345"));
params.add(new BasicNameValuePair("param-2", "Hello!"));
httppost.setEntity(new UrlEncodedFormEntity(params, "UTF-8"));

//Execute and get the response.
HttpResponse response = httpclient.execute(httppost);
HttpEntity entity = response.getEntity();

if (entity != null) {
    try (InputStream instream = entity.getContent()) {
        // do something useful
    }
}
```

HTTP Client example > java 9

```
byte[] sampleData = "Sample request body".getBytes();
HttpRequest request = HttpRequest.newBuilder()
  .uri(new URI("https://postman-echo.com/post"))
  .headers("Content-Type", "text/plain;charset=UTF-8")
  .POST(HttpRequest.BodyProcessor.fromByteArray(sampleData))
  .build();
```

```
HttpResponse<String> response = HttpClient.newBuilder()
  .followRedirects(HttpClient.Redirect.ALWAYS)
  .build()
  .send(request, HttpResponse.BodyHandler.asString());
```

Integrate the Payment service

- Using native java http client update the existing payment service from the main application (Shopify) to make http request to the payment service

Feign

Introduction to Feign

- Feign makes writing web service clients easier with pluggable annotation support, which includes Feign annotations and JAX-RS annotations.
- A great thing about using Feign is that we don't have to write any code for calling the service, other than an interface definition.

Dependency

- Require spring-cloud.

```
<dependency>
    <groupId>org.springframework.cloud</groupId>
    <artifactId>spring-cloud-starter-openfeign</artifactId>
</dependency>
```

```
 <dependencyManagement>
     <dependencies>
         <dependency>
             <groupId>org.springframework.cloud</groupId>
             <artifactId>spring-cloud-dependencies</artifactId>
             <version>${spring-cloud.version}</version>
             <type>pom</type>
             <scope>import</scope>
         </dependency>
     </dependencies>
</dependencyManagement>
```

Configuration Feign

- Enable fain client
  - `@EnableFeignClients`
- Configuration can be done using
  - Custom Bean Configuration
  - Application properties

Example

```
@FeignClient(value = "jplaceholder", url = "https://jsonplaceholder.typicode.com/")
public interface JSONPlaceHolderClient {

    @RequestMapping(method = RequestMethod.GET, value = "/posts")
    List<Post> getPosts();

    @RequestMapping(method = RequestMethod.GET, value = "/posts/{postId}", produces = "application/json")
    Post getPostById(@PathVariable("postId") Long postId);
}
```

Configurations

- Setting URI
- Specifying the HTTP Method
- Setting Headers
- Setting a Timeout
- Setting a Request Body

/ Error Handling for REST with Spring

Definition

- Error handling refers to the anticipation, detection, and resolution of programming, application, and communications errors. Specialized programs, called error handlers, are available for some applications. The best programs of this type forestall errors if possible, recover from them when they occur without terminating the application, or (if all else fails) gracefully terminate an affected application and save the error information to a log file.

Controller-Level

- The first solution works at the @Controller level. We will define a method to handle exceptions and annotate that with @ExceptionHandler:

```
public class FooController{

    //...
    @ExceptionHandler({ CustomException1.class, CustomException2.class })
    public void handleException() {
        //
    }
}
```

Drawback

- The @ExceptionHandler annotated method is only active for that particular Controller, not globally for the entire application.
- Of course, adding this to every controller makes it not well suited for a general exception handling mechanism.
- We can work around this limitation by having all Controllers extend a Base Controller class.

HandlerExceptionResolver

- This will resolve any exception thrown by the application. It will also allow us to implement a uniform exception handling mechanism in our REST API.
- This resolver was introduced in Spring 3.0, and it's enabled by default in the DispatcherServlet.

HandlerExceptionResolver Example

```
@ResponseStatus(value = HttpStatus.NOT_FOUND)
public class MyResourceNotFoundException extends RuntimeException {
    public MyResourceNotFoundException() {
        super();
    }
    public MyResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
    public MyResourceNotFoundException(String message) {
        super(message);
    }
    public MyResourceNotFoundException(Throwable cause) {
        super(cause);
    }
}
```

ResponseStatusException

```
@GetMapping(value = "/{id}")
public Foo findById(@PathVariable("id") Long id, HttpServletResponse response) {
    try {
        Foo resourceById = RestPreconditions.checkFound(service.findOne(id));

        eventPublisher.publishEvent(new SingleResourceRetrievedEvent(this, response));
        return resourceById;
     }
    catch (MyResourceNotFoundException exc) {
         throw new ResponseStatusException(
           HttpStatus.NOT_FOUND, "Foo Not Found", exc);
    }
}
```

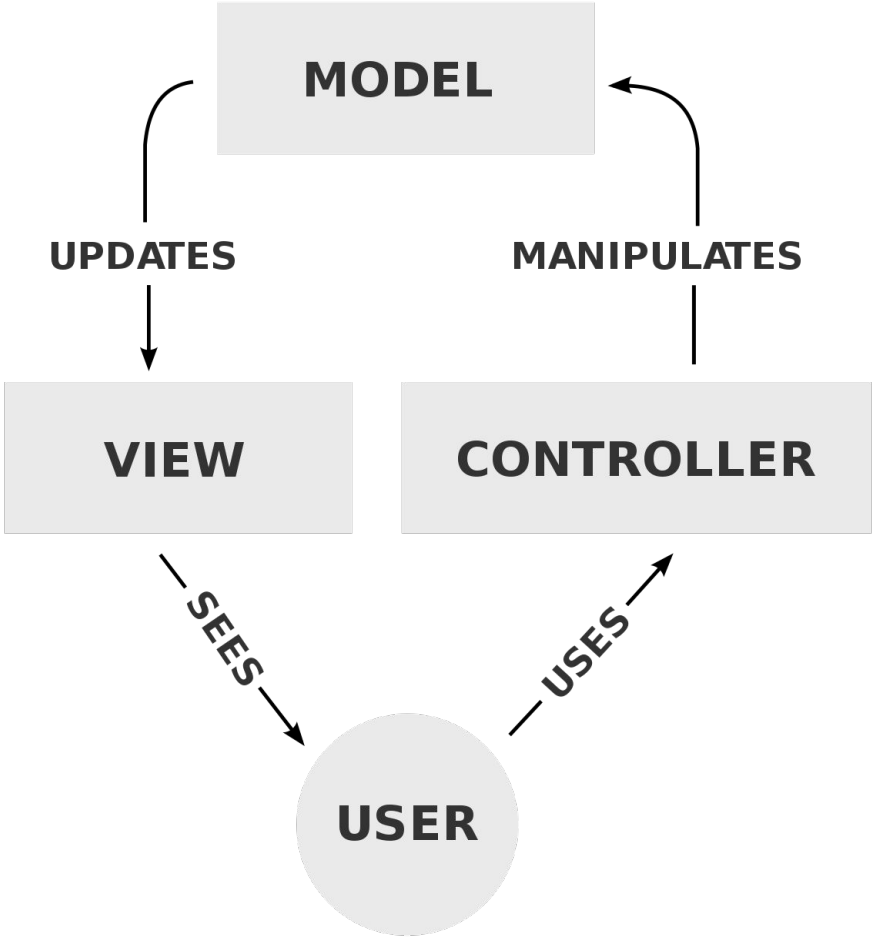
REST and Method-Level Security

```
@ControllerAdvice
public class RestResponseEntityExceptionHandler
  extends ResponseEntityExceptionHandler {

    @ExceptionHandler({ AccessDeniedException.class })
    public ResponseEntity<Object> handleAccessDeniedException(
      Exception ex, WebRequest request) {
        return new ResponseEntity<Object>(
          "Access denied message here", new HttpHeaders(), HttpStatus.FORBIDDEN);
    }

    ...
}
```

MVC

Definition

- Model–view–controller (MVC) is a software design pattern commonly used for developing user interfaces that divide the related program logic into three interconnected elements.
- This is done to separate internal representations of information from the ways information is presented to and accepted from the user.
- Traditionally used for desktop graphical user interfaces (GUIs), this pattern became popular for designing web applications.
- Popular programming languages have MVC frameworks that facilitate implementation of the pattern.


MODEL
UPDATES
MANIPULATES
VIEW
CONTROLLER
SEES
USES
USER

Components

- The Model encapsulates the application data and in general they will consist of POJO.

- The View is responsible for rendering the model data and in general it generates HTML output that the client's browser can interpret.

- The Controller is responsible for processing user requests and building an appropriate model and passes it to the view for rendering.

```java
@Controller
public class HelloController {
   @RequestMapping(value = "/hello", method = RequestMethod.GET)
   public String printHello(ModelMap model) {
      model.addAttribute("message", "Hello Spring MVC Framework!");
      return "hello";
   }
}
```

Monolithic Architecture

Advantages of monolithic

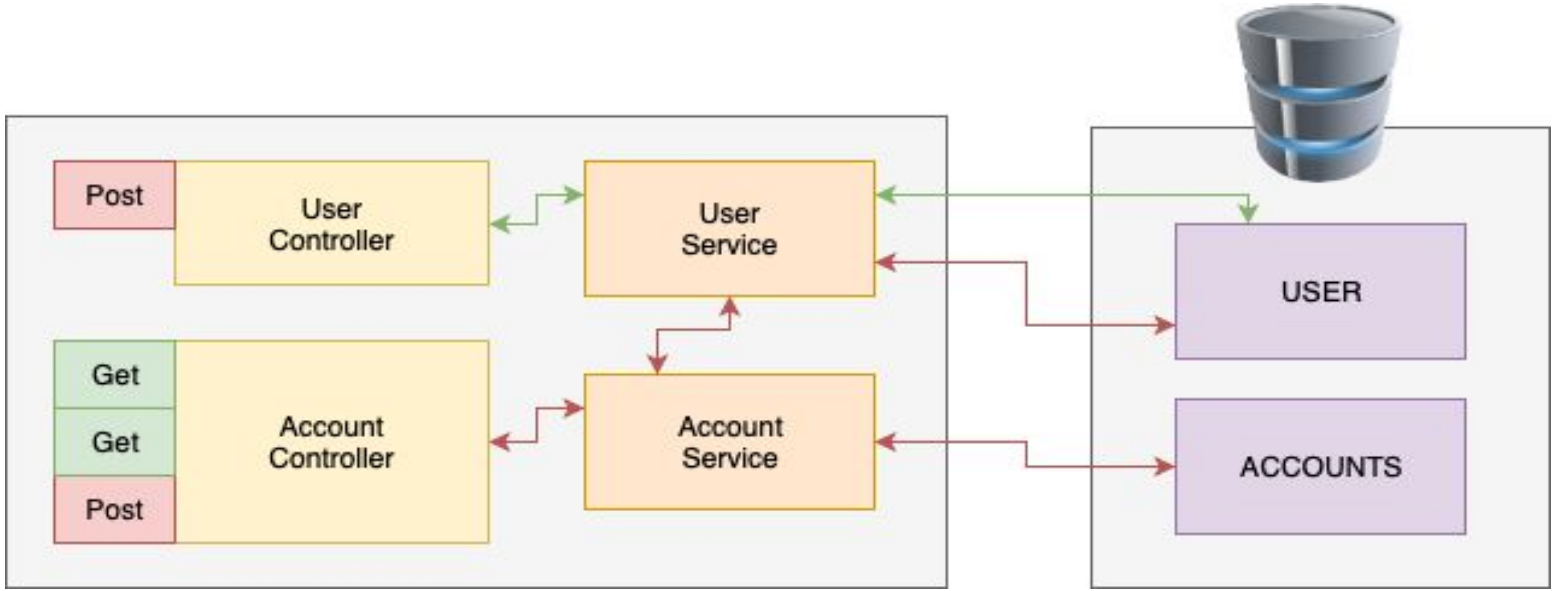
- Simple to develop
- Simple to test
  - For example you can implement end-to-end testing by simply launching the application and testing the UI with Selenium
- Simple to deploy.
  - You just have to copy the packaged application to a server
- Simple to scale horizontally by running multiple copies behind a load balancer

Drawbacks of Monolithic Architecture

- This simple approach has a limitation in size and complexity
- Application is too large and complex to fully understand and made changes fast and correctly
- The size of the application can slow down the start-up time
- You must redeploy the entire application on each update
- Impact of a change is usually not very well understood which leads to do extensive manual testing
- Continuous deployment is difficult

Drawbacks of Monolithic Architecture

- Monolithic applications can also be difficult to scale when different modules have conflicting resource requirements
- Another problem with monolithic applications is reliability. Bug in any module (e.g. memory leak) can potentially bring down the entire process. Moreover, since all instances of the application are identical, that bug will impact the availability of the entire application
- Monolithic applications has a barrier to adopting new technologies. Since changes in frameworks or languages will affect an entire application it is extremely expensive in both time and cost


Post
User Controller
User Service
Get
Get
Post
Account Controller
Account Service
USER
ACCOUNTS

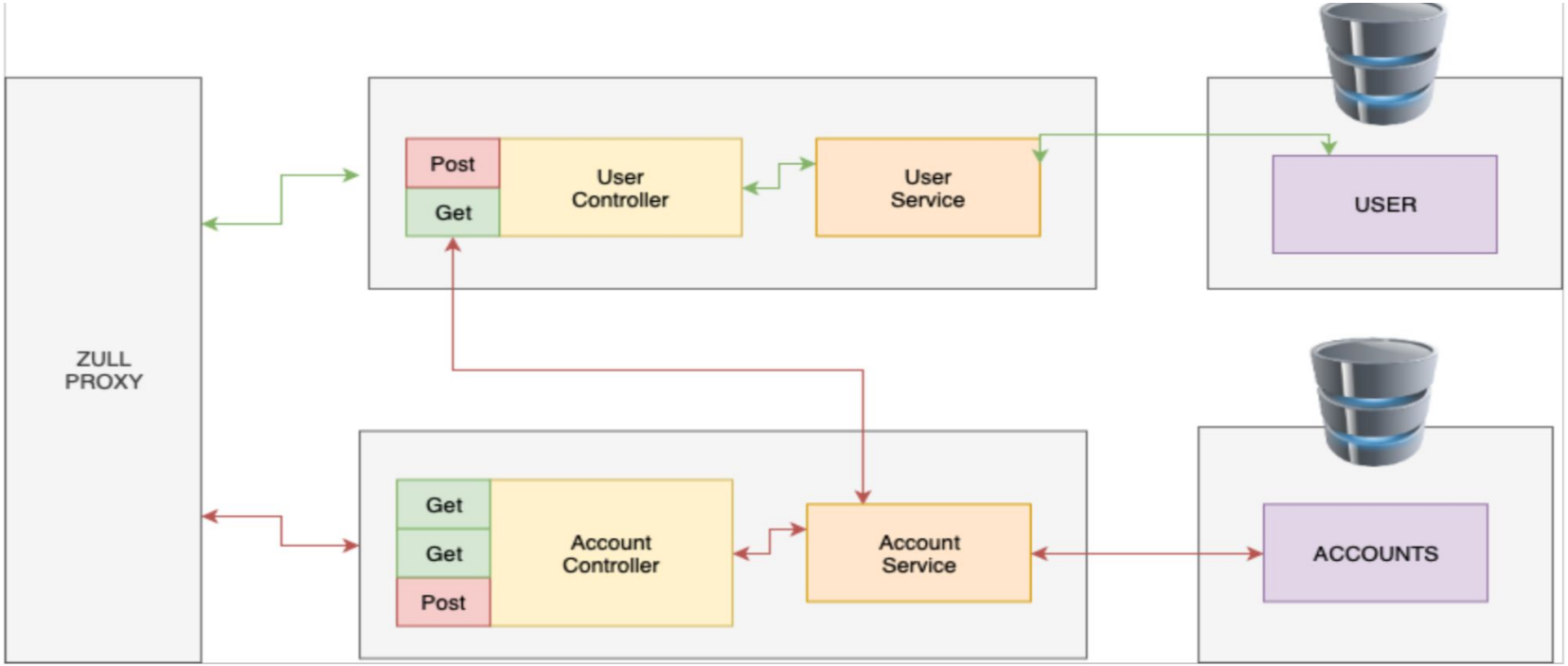
Microservice Architecture

Why to migrate to micro-services

- Because you „need to be in Cloud”
- Developers think is cool
- Manager thinks is good point for sales

Advantages of micro-services

- Modelled around business domain
- Hide Implementation details
- Decentralise all the Things
- Deploy independently
- Isolate Failure
- Highly observable


ZULL
PROXY
Post
Get
User
Controller
User
Service
USER
Get
Get
Post
Account
Controller
Account
Service
ACCOUNTS

OpenApi Generator Server

OpenApi Generator

- As the name suggests, the OpenAPI Generator generates code from an OpenAPI specification.
- It can create code for client libraries, server stubs, documentation, and configuration. It supports various languages and frameworks.
- Notably, there's support for C++, C#, Java, PHP, Python, Ruby, Scala – almost all the widely used ones.

Dependency

```
<!--openApi dependency-->
<dependency>
        <groupId>org.openapitools</groupId>
        <artifactId>jackson-databind-nullable</artifactId>
        <version>0.2.1</version>
</dependency>
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger2</artifactId>
        <version>3.0.0</version>
</dependency>
<dependency>
        <groupId>javax.validation</groupId>
        <artifactId>validation-api</artifactId>
        <version>2.0.1.Final</version>
</dependency>

<!--swagger ui-->
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-boot-starter</artifactId>
        <version>3.0.0</version>
</dependency>
<dependency>
        <groupId>io.springfox</groupId>
        <artifactId>springfox-swagger-ui</artifactId>
        <version>3.0.0</version>
</dependency>
```

Plugin

```xml
<plugin>
    <groupId>org.openapitools</groupId>
    <artifactId>openapi-generator-maven-plugin</artifactId>
    <version>4.3.1</version>
    <executions>
        <execution>
            <goals>
                <goal>generate</goal>
            </goals>
            <configuration>
                <inputSpec>${project.basedir}/api/server.yaml</inputSpec>
                <generatorName>spring</generatorName>
                <apiPackage>com.rd.demo.server.server.api</apiPackage>
                <modelPackage>com.rd.demo.server.server.model</modelPackage>
                <supportingFilesToGenerate>ApiUtil.java</supportingFilesToGenerate>
                <configOptions>
                    <delegatePattern>true</delegatePattern>
                    <useTags>true</useTags>
                </configOptions>
            </configuration>
        </execution>
    </executions>
</plugin>
```

How to use

- After a maven clean install the server side controller will be generated
- Inside target/{apiPackage} can be found the generated controller
- They can be enabled inside application extending delegated classes inside a service bean

Q&A


OBILE/ACADEMY