# BUILDING REST WITH SPRING

banuprakashc@yahoo.co.in

# Agenda

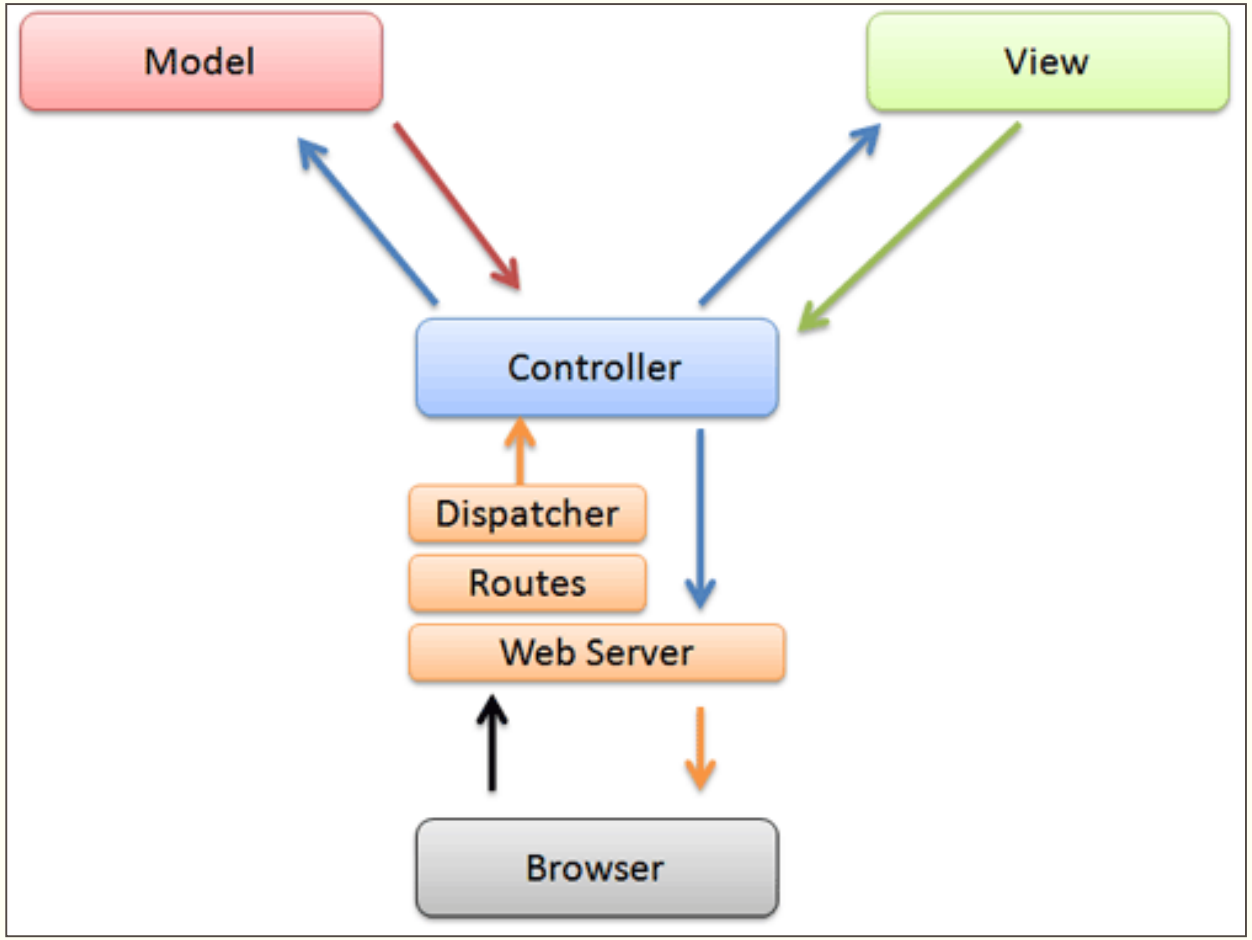
- Understand Spring MVC
- Rest in Spring
- The Java Configuration
- MessageConvertors
- @ResponseBody and @RequestBody
- Custom Message convertors
- ErrorHandling
- RestTemplate
- Testing
- HATEOAS
- Security
- Documentation

# Model View Controller (MVC)

- MVC = Model-View-Controller
  - Clearly separates business, navigation and presentation logic
  - Proven mechanism for building a thin and clean web-tier.
- Three core collaborating components
  - Controller
    - Handles navigation logic and interacts with the service tier for business logic
  - Model
    - The contract between the Controller and the View
    - Contains the data needed to render the View
    - Populated by the Controller
  - View
    - Renders the response to the request
    - Pulls data from the model

# Model View Controller (MVC)

- MVC Components



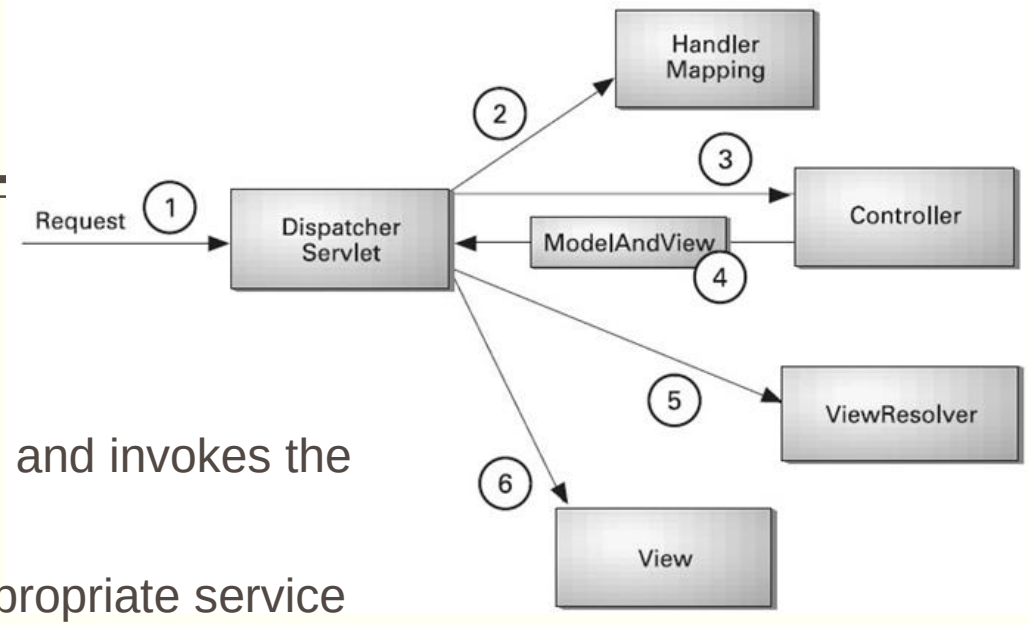
# Model View Controller (MVC)

- Motivation
  - Eases maintenance burden
    - Changes to business logic are less likely to break the presentation logic
    - Changes to presentation logic  also does not break business logic.
  - Facilitates multi-disciplined team development
    - Developers can focus on creating robust business code without having to worry about breaking the UI
    - Designers can focus on building usable and engaging UIs without worrying about Java
  - Use the best tool for the job
    - Java is especially suited to creating business logic code
    - Markup or template languages are more suited to creating HTML layouts.
  - Ease testability
    - Business and navigation logic are separated from presentation logic meaning they can be tested separately
    - Practically: you can test more code outside the Servlet container

# Spring MVC

- Core Components of Spring MVC
  - DispatcherServlet
    - Spring’s Front Controller implementation. Request routing is completely controlled by the Front Controller. As an application developer, you will have to just configure the DispatcherServlet in web.xml
  - Controller
    - An application developer created component for handling requests.
    - Controllers are POJOs which are managed by Spring ApplicationContext just like any other bean
    - Controllers encapsulates navigation logic.
  - View
    - An application developer created pages responsible for rendering output.

# Spring MVC



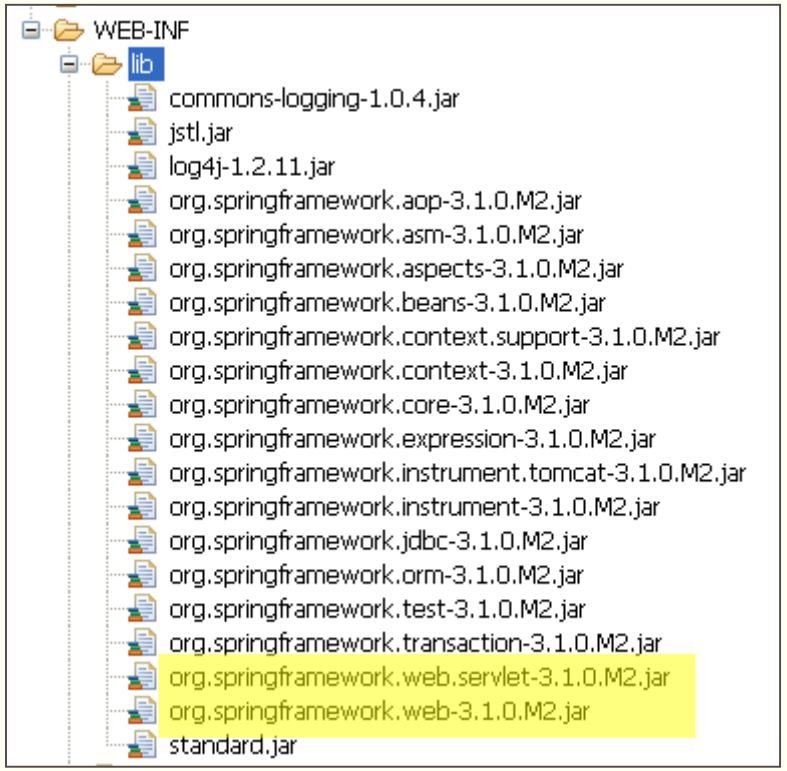

- The DispatcherServlet first receives the request
- The DispatcherServlet consults the HandlerMapping and invokes the Controller associated with the request
- The Controller process the request by calling the appropriate service methods
- The Controller returns a ModeAndView object to the DispatcherServlet. The ModeAndView object contains the model data and the view name.
- The DispatcherServlet sends the view name to a ViewResolver to find the actual View to invoke.
- Now the DispatcherServlet will pass the model object to the View to render the result.The View with the help of the model data will render the result back to the user

# Spring 3 MVC

- Why Spring 3 MVC?
  - Spring 3 introduces a mvc namespace that greatly simplifies Spring MVC setup.
    - Using mvc namespace Controllers, ViewResoulvers, interceptors and resources configuration becomes that much easier.
  - No changes to the DispatcherServlet configuration in web.xml
  - Many other enhancements makes it easier to get Spring 3.x web applications up and running.

# Spring 3 MVC

- Step 1
  - Create a Dynamic Web Project
  - Add Spring 3 Libraries to WEB-INF/lib folder




Spring libraries required for building Spring MVC application

# Spring 3 MVC

- Step 2
  - Configure DispatcherServlet in web.xml [ this remains the same for every version of spring MVC application]

```
<servlet>
    <servlet-name>dispatcher</servlet-name>
    <servlet-class>org.springframework.web.servlet.DispatcherServlet</servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>

<servlet-mapping>
    <servlet-name>dispatcher</servlet-name>
    <url-pattern>/</url-pattern>
</servlet-mapping>
```

  - The DispatcherServlet is configured as the default Servlet for the application (mapped to "/")

# Spring 3 MVC

- Step 3 Configure Controllers and View Resolver

```
<beans xmlns="http://www.springframework.org/schema/beans"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:mvc="http://www.springframework.org/schema/mvc"
    xmlns:context="http://www.springframework.org/schema/context"
    xsi:schemaLocation="http://www.springframework.org/schema/beans
         http://www.springframework.org/schema/beans/spring-beans-3.0.xsd
         http://www.springframework.org/schema/context
         http://www.springframework.org/schema/context/spring-context-3.0.xsd
         http://www.springframework.org/schema/mvc
         http://www.springframework.org/schema/mvc/spring-mvc-3.0.xsd">

    <!-- Configures the @Controller programming model -->
    <mvc:annotation-driven />
    <context:component-scan base-package=" com.banu.controllers    "/>

    <!-- Forwards requests to the "/" resource to the "home" view -->
    <mvc:view-controller path="/" view-name="home" />

    <!-- Resolves view names to protected ".jsp" within the /WEB-INF/pages directory  -->
    <bean id="viewResolver"
        class=" org.springframework.web.servlet.view.InternalResourceViewResolver">
        <property name="prefix" value="/WEB-INF/pages/" />
        <property name="suffix" value=".jsp" />
    </bean>
</beans>
```

Registers the HandlerMapping required to dispatch requests to your @Controllers

# Spring 3 MVC

- Step 4
  - Coding your first controller using annotations

```
import java.util.Date;

import org.springframework.stereotype.Controller;
import org.springframework.web.bind.annotation.RequestMapping;
import org.springframework.web.servlet.ModelAndView;

/**
 * @author Banu Prakash
 *
 */
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public ModelAndView helloWorld() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("helloWorld");
        mav.addObject("message", "Hello World!");
        mav.addObject("time", new Date());
        return mav;
    }
}
```

@Controller annotation allows for auto detection of Controller

@RequestMapping("path") specifies that the method is invoked to handle the request path.

# Spring 3 MVC

- Step 5
  - Home Page

http://localhost:8080/Spring3MVC/

Say Hello World

(1)

WEB-INF/pages/home.jsp

```
<body>
    <a href="helloWorld"> Say Hello World</a>
</body>
```

(2)

```
@Controller
public class HelloWorldController {

    @RequestMapping("/helloWorld")
    public ModelAndView helloWorld() {
        ModelAndView mav = new ModelAndView();
        mav.setViewName("helloWorld");
        mav.addObject("message", "Hello World!");
        mav.addObject("time", new Date());
        return mav;
    }
}
```

(3)

WEB-INF/pages/helloWorld.jsp

```
<body>
    Message : ${requestScope.message} <br />
    Time : ${requestScope.time}
</body>
```

Rendered output

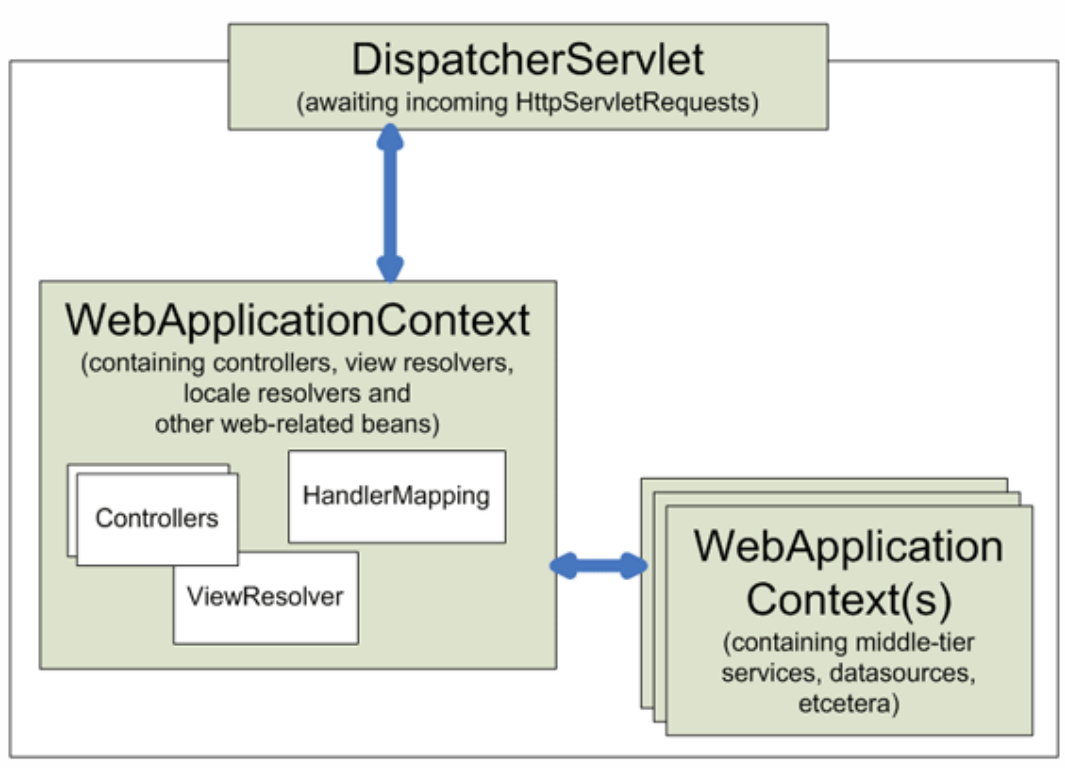
Hello World View

http://localhost:8080/Spring3MVC/helloWorld

Message : Hello World!
Time : Thu Jul 21 11:10:37 IST 2011

# Spring MVC - Business Layer integration

- The WebApplicationContext is an extension of the plain ApplicationContext that has some extra features necessary for web applications



# Spring MVC - Business Layer integration

- ContextLoaderListener a servlet listener which is responsible for loading additional WebApplicationContext mostly consisting of beans for service layer and dao layer.

```
<listener>
    <listener-class>org.springframework.web.context.ContextLoaderListener</listener-class>
</listener>

<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>/WEB-INF/beans.xml</param-value>
</context-param>
```

beans.xml file consists of all the Dao/Service Layer configurations as discussed in Hibernate Spring integration

- The ContextLoaderListener looks for /WEB-INF/applicationContext.xml by default, but you can override it using the context parameter contextConfigLocation as shown.

# REST in Spring

- The Spring framework supports 2 ways of creating RESTful services:
  - **MVC with ModelAndView**
    - The ModelAndView approach is older and much better documented, but also more verbose and configuration heavy.
    - It tries to integrate the REST paradigm into the old model, which is not without problems.
  - Using **HTTP message converters**
    - The Spring team understood this and provided first-class REST support starting with Spring 3.0.
    - The new approach, based on HttpMessageConverter and annotations, is much more lightweight and easy to implement.
    - Configuration is minimal and it provides sensible defaults for what you would expect from a RESTful service.
    - It is however newer and a bit on the light side concerning documentation;.
    - Nevertheless, this is the way RESTful services should be build after Spring 3.0.

# The Maven pom.xml

```xml
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-webmvc</artifactId>
    <version>4.1.4.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-web</artifactId>
    <version>4.1.4.RELEASE</version>
</dependency>

<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-core</artifactId>
    <version>4.1.4.RELEASE</version>
</dependency>

<dependency>
    <groupId>javax.servlet</groupId>
    <artifactId>javax.servlet-api</artifactId>
    <version>3.1.0</version>
    <scope>provided</scope>
</dependency>
```

# Additional Maven dependencies

- In addition to the spring-webmvc dependency required for the standard web application, we'll need to set up content marshalling and unmarshalling for the REST API:
  - These are the libraries used to convert the representation of the REST resource to either JSON or XML.

```
<!-- Jackson JSON Processor -->
<dependency>
    <groupId>com.fasterxml.jackson.core</groupId>
    <artifactId>jackson-databind</artifactId>
    <version>2.4.1</version>
</dependency>
<dependency>
    <groupId>com.thoughtworks.xstream</groupId>
    <artifactId>xstream</artifactId>
    <version>1.4.8</version>
</dependency>
<dependency>
    <groupId>org.springframework</groupId>
    <artifactId>spring-oxm</artifactId>
    <version>3.0.6.RELEASE</version>
</dependency>
```

# The Java Configuration

- Setting up the FrontController- DispatcherServlet

```java
public class AppInitializer implements WebApplicationInitializer {

    @Override
    public void onStartup(ServletContext servletContext)
            throws ServletException {
        WebApplicationContext context = getContext();
        servletContext.addListener(new ContextLoaderListener(context));
        ServletRegistration.Dynamic dispatcher = servletContext.addServlet(
                "DispatcherServlet", new DispatcherServlet(context));
        dispatcher.setLoadOnStartup(1);
        dispatcher.addMapping("/rest/*");
    }

    private WebApplicationContext getContext() {
        AnnotationConfigWebApplicationContext context = new AnnotationConfigWebApplicationContext();
        context.setConfigLocation("com.sample");
        return context;
    }

}
```

# The Java Configuration

- The new @EnableWebMvc annotation
  - Used to detect the existence of Jackson and JAXB 2 on the classpath and automatically creates and registers default JSON and XML converters.
  - The functionality of the annotation is equivalent to the XML version:

    <mvc:annotation-driven />

```
@EnableWebMvc
@Configuration
@ComponentScan({ "com.sample" })
public class WebConfig extends WebMvcConfigurerAdapter {

}
```

# Message Converters

- By default, the following HttpMessageConverters instances are pre-enabled:
  - ByteArrayHttpMessageConverter – converts byte arrays
  - StringHttpMessageConverter – converts Strings
  - ResourceHttpMessageConverter – converts org.springframework.core.io.Resource for any type of octet stream
  - FormHttpMessageConverter – converts form data to/from a MultiValueMap<String, String>.
  - **Jaxb2RootElementHttpMessageConverter** – converts Java objects to/from XML (added only if JAXB2 is present on the classpath)
  - **MappingJackson2HttpMessageConverter** – converts JSON (added only if Jackson 2 is present on the classpath)
  - **MappingJacksonHttpMessageConverter** – converts JSON (added only if Jackson is present on the classpath)
  - AtomFeedHttpMessageConverter and RssChannelHttpMessageConverter – converts Atom/RSS feeds (added only if Rome is present on the classpath)

# Message Converters

- HttpMessageConverter implementation has one or several associated MIME Types.
- Receiving a new request
  - Spring will use of the “Accept” header to determine the media type that it needs to respond with.
  - It will then try to find a registered converter that is capable of handling that specific media type – and it will use it to convert the entity and send back the response.
- Receiving a new request which contains JSON information
  - the framework will use the “Content-Type” header to determine the media type of the request body.
  - It will then search for a HttpMessageConverter that can convert the body sent by the client to a Java Object.

# @ResponseBody

- @ResponseBody on a Controller method indicates to Spring that the return value of the method is serialized directly to the body of the HTTP Response.
- As discussed the “Accept” header specified by the Client will be used to choose the appropriate Http Converter to marshall the entity.

```
@RequestMapping(value = "/products", method = RequestMethod.GET)
public @ResponseBody List<Product> getProducts() {
    return orderService.getProducts();
}

@RequestMapping(value = "/orders/{id}",
        method = RequestMethod.GET)
public @ResponseBody List<Order> getOrders(@PathVariable("id")int id) {
    return orderService.getOrders(id);
}
```

# @RequestBody

- @RequestBody is used on the argument of a Controller method – it indicates to Spring that the body of the HTTP Request is deserialized to that particular Java entity.
- As discussed “Content-Type” header specified by the Client will be used to determine the appropriate converter for this.

```
@RequestMapping(value = "/customers", method = RequestMethod.POST)
public @ResponseBody List<Customer> add(@RequestBody Customer customer) {
    orderService.addCustomer(customer);
    return orderService.getAllCustomers();
}
```

# Custom Converters Configuration

- We can customize the message converters by extending the WebMvcConfigurerAdapter class and overriding the configureMessageConverters method
  - By extending this support class, we are losing the default message converters which were previously pre-registered – we only have what we define  XStream library now needs to be present onMarshallingHttpMessageConverter – and we're using the Spring XStream support to configure it.
- This allows a great deal of flexibility since we're working with the low level APIs of the underlying marshalling framework – in this case XStream – and we can configure that however we want.
- We can of course now do the same for Jackson – by defining our own MappingJackson2HttpMessageConverter we can now set a custom ObjectMapper on this converter and have it configured as we need to.
- In this case XStream was the selected marshaller/unmarshaller implementation, but others like CastorMarshaller can be used to – refer to Spring api documentation for full list of available marshallers. the classpath.

# Custom Converters Configuration

```
@EnableWebMvc
@Configuration
@ComponentScan({ "com.sample" })
public class WebConfig extends WebMvcConfigurerAdapter {

    @Override
    public void configureContentNegotiation(
            ContentNegotiationConfigurer configurer) {
        configurer.favorPathExtension(true).
        favorParameter(true).
        parameterName("mediaType").
        ignoreAcceptHeader(true).useJaf(false)
                .defaultContentType(MediaType.APPLICATION_JSON)
                .mediaType("xml", MediaType.APPLICATION_XML)
                .mediaType("json", MediaType.APPLICATION_JSON);
    }
```

# Custom Converters Configuration

- Jackson and XStream libraries now needs to be present on the classpath.

```
@Override
public void configureMessageConverters(
        List<HttpMessageConverter<?>> converters) {
    converters.add(createXmlHttpMessageConverter());
    MappingJackson2HttpMessageConverter jsonMapper = new MappingJackson2HttpMessageConverter();
    jsonMapper.setPrettyPrint(true);
    converters.add(jsonMapper);

    super.configureMessageConverters(converters);
}

private HttpMessageConverter<Object> createXmlHttpMessageConverter() {
    MarshallingHttpMessageConverter xmlConverter = new MarshallingHttpMessageConverter();
    // CastorMarshaller marshaller = new CastorMarshaller();
    XStreamMarshaller xstreamMarshaller = new XStreamMarshaller();
    Map map = new HashMap<>();
    map.put("customer", com.sample.rest.entity.Customer.class);
    try {
        xstreamMarshaller.setAliases(map);
    } catch (ClassNotFoundException e) {
        e.printStackTrace();
    }
    xmlConverter.setMarshaller(xstreamMarshaller);
    xmlConverter.setUnmarshaller(xstreamMarshaller);
    return xmlConverter;
}
```

# ResponseEntity<T>

- HttpEntity<T>: Represents an HTTP request or response entity, consisting of headers and body.
- **ResponseEntity<T>:** Extension of HttpEntity that adds a HttpStatus status code.

```
@RequestMapping("/handle")
 public ResponseEntity<String> handle() {
   HttpHeaders responseHeaders = new HttpHeaders();
   responseHeaders.set("MyResponseHeader", "MyValue");
   return new ResponseEntity<String>("Hello World", responseHeaders, HttpStatus.CREATED);
 }
```

# Error Handling

- The communication aspect is especially important to developers consuming a REST API.
- Well-designed error responses allow consuming developers to understand the issues and help them to use the API correctly. Additionally, good error handling allows API developers to log information that can aid in debugging issues on their end

# Custom exception

- @ResponseStatus annotation is declared at the class level. The annotation instructs Spring MVC that an HttpStatus NOT_FOUND (404 code) should be used in the response when a ResourceNotFoundException is thrown.

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

    private static final long serialVersionUID = 1L;

    public ResourceNotFoundException() {}

    public ResourceNotFoundException(String message) {
        super(message);
    }

    public ResourceNotFoundException(String message, Throwable cause) {
        super(message, cause);
    }
}
```

# Using Custom exception

- Throw ResourceNotFoundException for non-existing resource

```
@RequestMapping(value = "/customers/{id}", method = RequestMethod.GET)
public @ResponseBody Customer getCustomer(@PathVariable("id") int id) {
    Customer customer =  orderService.getCustomer(id);
        if(customer == null) {
            throw new ResourceNotFoundException("Customer with id " + id + " not found");
        }
    return customer;
}
```

# Error Handling for REST with Spring

- **The Controller level *@ExceptionHandler***
  - @ExceptionHandler annotated method is only active for that particular Controller, not globally for the entire application

```
@ExceptionHandler({ResourceNotFoundException.class})
@ResponseStatus(value=HttpStatus.NOT_FOUND)
@ResponseBody
public ErrorInfo handleTypeMismatchException(HttpServletRequest req, ResourceNotFoundException ex) {
    String errorURL = req.getRequestURL().toString();
    return new ErrorInfo(errorURL, ex.getMessage(), 404);
}
```

localhost:8080/restapp/rest/customers/10

```
{
  "url" : "http://localhost:8080/restapp/rest/customers/10",
  "message" : "Customer with id 10 not found",
  "status" : 404
}
```

# @ControllerAdvice

- Error handling is a crosscutting concern. We need an application-wide strategy that handles all of the errors in the same way and writes the associated details to the response body
- @ControllerAdvice can be used to implement such crosscutting concerns

```
@ControllerAdvice
public class RestExceptionHandler {

        @ExceptionHandler(ResourceNotFoundException.class)
        public ResponseEntity<?> handleResourceNotFoundException(ResourceNotFoundException
        rnfe, HttpServletRequest request) {

                ErrorDetail errorDetail = new ErrorDetail();
                errorDetail.setTimeStamp(new Date().getTime());
                errorDetail.setStatus(HttpStatus.NOT_FOUND.value());
                errorDetail.setTitle("Resource Not Found");
                errorDetail.setDetail(rnfe.getMessage());
                errorDetail.setDeveloperMessage(rnfe.getClass().getName());

                return new ResponseEntity<>(errorDetail, null, HttpStatus.NOT_FOUND);
        }

}
```

# RestTemplate

- The RestTemplate is the central Spring class for client-side HTTP access.
- RestTemplate is thread-safe once constructed, and that you can use callbacks to customize its operations.
- **RestTemplate Methods**

| HTTP | METHOD |
|---|---|
| GET | getForObject(String, Class, String...) |
| POST | postForLocation(String, Object, String...) |
| PUT | put(String, Object, String...) |
| DELETE | delete(String, String...) |
| OPTIONS | optionsForAllow(String, String...) |
| HEAD | headForHeaders(String, String...) |

# RestTemplate

- GET

```
private static void getCustomers() {
    final String uri = "http://localhost:8080/restapp/rest/orders/100?mediaType=json";

    RestTemplate restTemplate = new RestTemplate();
    String result = restTemplate.getForObject(uri, String.class);

    System.out.println(result);
}
```


```
[ {
  "items" : [ {
    "product" : {
      "id" : 1,
      "name" : "L.G Washing Machine",
      "price" : 23000.9
    },
    "quantity" : 1,
    "price" : 23000.9
  }, {
    "product" : {
      "id" : 2,
      "name" : "MotoG Mobile",
      "price" : 12999.9
    },
    "quantity" : 2,
    "price" : 25999.8
  } ],
  "orderDate" : 1441018321536
} ]
```

# RestTemplate

- POST JSON

```
final String uri = "http://localhost:8080/restapp/rest/products.json";
RestTemplate restTemplate = new RestTemplate();
restTemplate.getMessageConverters().add(new MappingJackson2HttpMessageConverter());
HttpHeaders headers = new HttpHeaders();
headers.setAccept(Arrays.asList(MediaType.APPLICATION_JSON));
headers.setContentType(MediaType.APPLICATION_JSON);

Product product = new Product(111, "TEST", 4444.44);
HttpEntity<Product> entity = new HttpEntity<Product>(product,headers);
ResponseEntity<Product> response =
  restTemplate.exchange(uri, HttpMethod.POST, entity, Product.class,product);

System.out.println(response.getBody());
```

# RestTemplate

- POST xml

```
private static void createProductXML() {
    final String uri = "http://localhost:9999/restapp/rest/products.xml";
    RestTemplate restTemplate = new RestTemplate();
    restTemplate.setMessageConverters(getMessageConverters());
    HttpHeaders headers = new HttpHeaders();
    headers.setAccept(Arrays.asList(MediaType.APPLICATION_XML));
    headers.setContentType(MediaType.APPLICATION_XML);
    Product product = new Product(111, "TEST", 4444.44);
    HttpEntity<Product> entity = new HttpEntity<Product>(product, headers);
    ResponseEntity<Product> response = restTemplate.exchange(uri,
            HttpMethod.POST, entity, Product.class, product);

    System.out.println(response.getBody());
}

private static List<HttpMessageConverter<?>> getMessageConverters() {
    XStreamMarshaller marshaller = new XStreamMarshaller();
    MarshallingHttpMessageConverter marshallingConverter = new MarshallingHttpMessageConverter(
            marshaller);

    List<HttpMessageConverter<?>> converters = new ArrayList<HttpMessageConverter<?>>();
    converters.add(marshallingConverter);
    return converters;
}
```

# TESTING REST Services

- The SpringJUnit4ClassRunner adds Spring integration by performing activities such as loading application context, injecting autowired dependencies, and running specified test execution listeners.
- For Spring to load and configure an application context, it needs the locations of the XML context files or the names of the Java configuration classes.
- We typically use the @ContextConfiguration annotation to provide this information to the SpringJUnit4ClassRunner class.

# Unit Testing REST Controllers

- Setup for Unit testing

```
@RunWith(SpringJUnit4ClassRunner.class)
@ContextConfiguration(classes = { AppInitializer.class, WebConfig.class })
@WebAppConfiguration
public class ProductTest {

    @Autowired
    private WebApplicationContext wac;

    private MockMvc mockMvc;

    @Before
    public void setup() {
        mockMvc = MockMvcBuilders.webAppContextSetup(wac).build();
    }
```

# Unit Testing REST Controllers

- Testing the result using jsonPath

```
public static class TestUtil {
    public static final MediaType APPLICATION_JSON_UTF8 =
            new MediaType(MediaType.APPLICATION_JSON.getType(),
                    MediaType.APPLICATION_JSON.getSubtype(), Charset.forName("utf8"));
}

@Test
public void findProductById() throws Exception {
    MvcResult result = mockMvc.perform(get("/products/{id}", 1L))
            .andExpect(status().isOk())
            .andExpect(content().contentType(TestUtil.APPLICATION_JSON_UTF8))
            .andExpect(jsonPath("$.id", is(1)))
            .andExpect(jsonPath("$.name", is("L.G Washing Machine")))
            .andExpect(jsonPath("$.price", is(23000.90))).andReturn();

    String content = result.getResponse().getContentAsString();
    System.out.println(content);
}
```

# Unit Testing REST Controllers

```
@Test
public void getAllProducts() throws Exception {
    mockMvc.perform(get("/products"))
            .andExpect(status().isOk())
            .andExpect(content().contentType(TestUtil.APPLICATION_JSON_UTF8))
              .andExpect(jsonPath("$", hasSize(4)))
            .andExpect(jsonPath("$[0].id", is(1)))
            .andExpect(jsonPath("$[0].name", is("L.G Washing Machine")))
            .andExpect(jsonPath("$[0].price", is(23000.90)))
            .andExpect(jsonPath("$[1].id", is(2)))
            .andExpect(jsonPath("$[1].name", is("MotoG Mobile")))
            .andExpect(jsonPath("$[1].price", is(12999.90)));
}
```

# Unit Testing REST Controllers

```
@RequestMapping(value = "/products", method = RequestMethod.POST)
public ResponseEntity<String> addProduct(@RequestBody Product product ) {
    productService.addProduct(product);
    return new ResponseEntity<>("Product " + product.getName() + "added ", HttpStatus.CREATED);
}
```

```
@Test
public void addProduct() throws Exception {
    mockMvc.perform(
            post("/products").content(
                    new ObjectMapper().writeValueAsString(new Product(200,
                            "a", 100.00))).contentType(
                    TestUtil.APPLICATION_JSON_UTF8)).andExpect(
            status().isCreated());
}
```

# DISCOVERABILITY AND HATEOAS

- Discoverability of an API is a topic that doesn’t get enough well deserved attention, and as a consequence very few APIs get it right.
- It is also something that, if done correctly, can make the API not only RESTful and usable but also elegant.
- To understand discoverability you need to understand that constraint that is Hypermedia As The Engine Of Application State (HATEOAS); this constraint of a REST API is about full discoverability of actions/transitions on a Resource from Hypermedia (Hypertext really), as the only driver of application state.

# HATEOAS

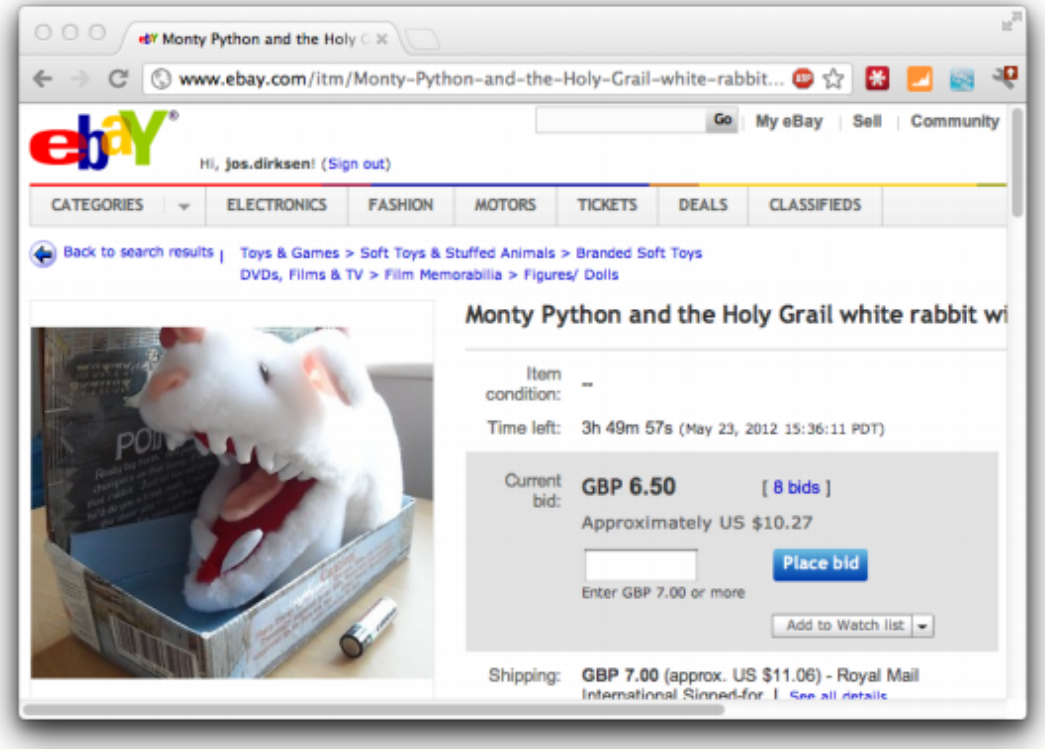
- HATEOAS - the word, there's no pronunciation for
- HATEOAS ("Hypermedia as the Engine of Application State") is an approach to building RESTful web services, where the client can dynamically discover the actions available to it at runtime from the server.
- The key to HATEOAS is simple
  - Hypermedia / Mime-types / Media-types :
    - Describes a current state
    - Compare it with a web page
    - Can be seen as the contract
  - Links:
    - Describe the transition to the next state
    - Compare it with hyperlinks
  - HATEOAS makes surfing the web possible

“In each response message, include the links for the next request message”

# HATEOAS

## Case: eBay

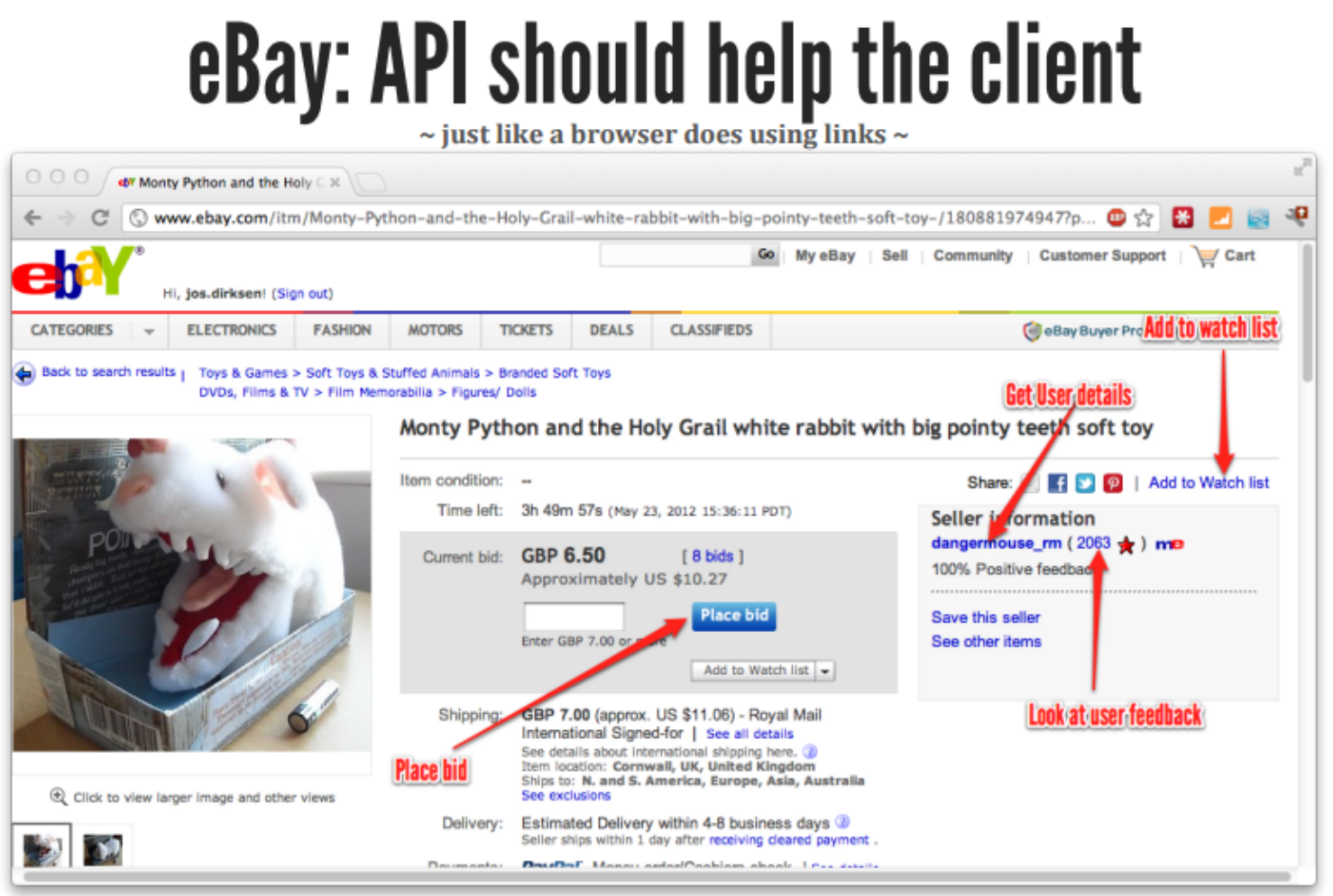
~ API should guide the user ~




### Common scenario: bidding on item

1. Add item to watch list: keep track of the item.
2. Get user details: find out more about the buyer.
3. Get user feedback: is seller trustworthy?
4. Make a bid: place a bid for the item.

# HATEOAS


eBay: API should help the client
~ just like a browser does using links ~
Monty Python and the Holy
www.ebay.com/itm/Monty-Python-and-the-Holy-Grail-white-rabbit-with-big-pointy-teeth-soft-toy-/180881974947?p...
Go
My eBay
Sell
Community
Customer Support
Cart
Hi, jos.dirksen! (Sign out)
CATEGORIES
ELECTRONICS
FASHION
MOTORS
TICKETS
DEALS
CLASSIFIEDS
eBay Buyer Pro
Add to watch list
Back to search results
Toys & Games > Soft Toys & Stuffed Animals > Branded Soft Toys
DVDs, Films & TV > Film Memorabilia > Figures/ Dolls
Get User details
Monty Python and the Holy Grail white rabbit with big pointy teeth soft toy
Item condition: --
Time left: 3h 49m 57s (May 23, 2012 15:36:11 PDT)
Share:
Add to Watch list
Current bid: GBP 6.50 [ 8 bids ]
Approximately US $10.27
Place bid
Enter GBP 7.00 or more
Add to Watch list
Seller information
dangerhouse_rm ( 2063 )
100% Positive feedback
Save this seller
See other items
Look at user feedback
Shipping: GBP 7.00 (approx. US $11.06) - Royal Mail International Signed-for | See all details
See details about international shipping here.
Item location: Cornwall, UK, United Kingdom
Ships to: N. and S. America, Europe, Asia, Australia
See exclusions
Place bid
Click to view larger image and other views
Delivery: Estimated Delivery within 4-8 business days
Seller ships within 1 day after receiving cleared payment .

# HATEOAS

## eBay: add to watchlist

~ API should tell us what to do ~

```
GET .../item/180881974947
{
 "name" : "Monty Python and the Holy Grail white rabbit big pointy teeth",
 "id" : "180881974947",
 "start-price" : "6.50",
 "currency" : "GBP",
 ...
 "links" : [
   { "type: "application/vnd.ebay.item",
     "rel": "Add item to watchlist",
     "href": "https://.../user/12345678/watchlist/180881974947"},
   {
     // and a whole lot of other operations
   ]
}
```

# HATEOAS

## eBay: get user details

~ Follow links to get more information ~

```
GET .../item/180881974947
{
 "name" : "Monty Python and the Holy Grail white rabbit big pointy teeth",
 "id" : "180881974947",
 "start-price" : "6.50",
 "currency" : "GBP",
 // whole lot of other general item data
 "bidder" : {
    "name" : "dangermouse_rm",
    "link" : {
      "type" : "application/vnd.ebay.user",
      "rel : "Get user details",
      "href : "https://.../user/314512346523"
    }
 }
}
```

# Spring HATEOAS

- Spring HATEOAS provides a set of useful types to ease working with those
- Links
  - The Link value object follows the Atom link definition and consists of a rel and an href attribute.
  - It contains a few constants for well known rels such as self, next etc.
  - The XML representation will render in the Atom namespace.

```
Link link = new Link("http://localhost:8080/something");
assertThat(link.getHref(), is("http://localhost:8080/something"));
assertThat(link.getRel(), is(Link.SELF));

Link link = new Link("http://localhost:8080/something", "my-rel");
assertThat(link.getHref(), is("http://localhost:8080/something"));
assertThat(link.getRel(), is("my-rel"));
```

# Spring HATEOAS

- Resources
  - Every representation of a resource will contain some links (at least the self one) spring provide a base class to actually inherit from when designing representation classes.
  - Inheriting from ResourceSupport will allow adding links easily

```
class PersonResource extends ResourceSupport {

  String firstname;
  String lastname;
}
```

```
PersonResource resource = new PersonResource();
resource.firstname = "Dave";
resource.lastname = "Matthews";
resource.add(new Link("http://myhost/people"));
```

JSON

```
{ firstname : "Dave",
  lastname : "Matthews",
  links : [ { rel : "self", href : "http://myhost/people" } ] }
```

XML

```
<person xmlns:atom="http://www.w3.org/2005/Atom">
  <firstname>Dave</firstname>
  <lastname>Matthews</lastname>
  <links>
    <atom:link rel="self" href="http://myhost/people" />
  </links>
</person>
```

# Spring HATEOAS

- ControllerLinkBuilder
  - Spring Hateoas provides a ControllerLinkBuilder that allows to create links by pointing to controller classes
  - The ControllerLinkBuilder uses Spring's ServletUriComponentsBuilder under the hood to obtain the basic URI information from the current request.
  - The ControllerLinkBuilder uses Spring's ServletUriComponentsBuilder under the hood to obtain the basic URI information from the current request. Assuming your application runs at http://localhost:8080/your-app
  - The builder now inspects the given controller class for its root mapping and thus end up with http://localhost:8080/your-app/people

```
import static org.sfw.hateoas.mvc.ControllerLinkBuilder.*;
```

```
Link link = linkTo(PersonController.class).withRel("people");
assertThat(link.getRel(), is("people"));
assertThat(link.getHref(), endsWith("/people"));
```

# Spring HATEOAS

- Build more nested links:

```
Person person = new Person(1L, "Dave", "Matthews");
//                  /person                 /     1
Link link = linkTo(PersonController.class).slash(person.getId()).withSelfRel();
assertThat(link.getRel(), is(Link.SELF));
assertThat(link.getHref(), endsWith("/people/1"));
```

# Spring HATEOAS

- Building links pointing to methods:
  - methodOn(…) creates a proxy of the controller class that is recording the method invocation and exposed it in a proxy created for the return type of the method

```
@RequestMapping(value = "/album/{id}", method = RequestMethod.GET)
@ResponseBody
public Resource<Album> getAlbum(@PathVariable(value = "id") String id) {
    Album album = musicService.getAlbum(id);
    return getAlbumResource(album);
}

private Resource<Album> getAlbumResource(Album album) {
    Resource<Album> resource = new Resource<Album>(album);
    // Link to Album
    resource.add(linkTo(methodOn(AlbumController.class).getAlbum(album.getId())).withSelfRel());
    // Link to Artist
    resource.add(linkTo(methodOn(ArtistController.class).getArtist(album.getArtist().getId())).withRel("artist"));
    // Option to purchase Album
    if (album.getStockLevel() > 0) {
        resource.add(linkTo(methodOn(AlbumController.class).purchaseAlbum(album.getId())).withRel("album.purchase"));
    }
    return resource;
}
```

# Example

- The entry point to our API is :
  - **http://localhost:8080/hateaos/rest/albums**
  - This will basically list all the albums available at our store, along with Stock Levels
  - It provides links to any client of our api telling it the URLs it can use to:
  - View and Albums details:
  - http://localhost:8080/hateaos/rest/album/{id}

```java
@RequestMapping(value = "/albums", method = RequestMethod.GET)
@ResponseBody
public Collection<Resource<Album>> getAllAlbums() {
    Collection<Album> albums = musicService.getAllAlbums();
    List<Resource<Album>> resources = new ArrayList<Resource<Album>>();
    for (Album album : albums) {
        resources.add(getAlbumResource(album));
    }
    return resources;
}
```


```json
[ {
  "id" : "3",
  "title" : "Like a Prayer",
  "artist" : {
    "id" : "MDNA",
    "name" : "madonna"
  },
  "stockLevel" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/hateaos/rest/album/3"
  }, {
    "rel" : "artist",
    "href" : "http://localhost:8080/hateaos/rest/artist/MDNA"
  } ]
}, {
  "id" : "2",
  "title" : "Thriller",
  "artist" : {
    "id" : "MJ",
    "name" : "Michael Jackson"
  },
  "stockLevel" : 3,
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/hateaos/rest/album/2"
  }, {
    "rel" : "artist",
    "href" : "http://localhost:8080/hateaos/rest/artist/MJ"
  }, {
    "rel" : "album.purchase",
    "href" : "http://localhost:8080/hateaos/rest/album/purchase/2"
  } ]
}, {
```

# Example

- View details of the Artist:
  - http://localhost:8080/hateaos/rest/artist/{id}
- Purchase a copy of the Album:
  - http://localhost:8080/hateaos/album/purchase/{id}

localhost:8080/hateaos/rest/album/purchase/1


```
{
  "id" : "1",
  "title" : "Billie Jean",
  "artist" : {
    "id" : "MJ",
    "name" : "Michael Jackson"
  },
  "stockLevel" : 0,
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/hateaos/rest/album/1"
  } ]
}
```


localhost:8080/hateaos/rest/album/4


```
{
  "id" : "4",
  "title" : "True Blue",
  "artist" : {
    "id" : "MDNA",
    "name" : "madonna"
  },
  "stockLevel" : 1,
  "links" : [ {
    "rel" : "self",
    "href" : "http://localhost:8080/hateaos/rest/album/4"
  }, {
    "rel" : "artist",
    "href" : "http://localhost:8080/hateaos/rest/artist/MDNA"
  }, {
    "rel" : "album.purchase",
    "href" : "http://localhost:8080/hateaos/rest/album/purchase/4"
  } ]
}
```

# Documenting REST Services

- Documentation is an important aspect of any project.
- This is especially true for enterprise and open source projects, where many people collaborate to build the project
- Documenting a REST API for consumers to use and interact with is a difficult task because there are no real established standards.
- Organizations have historically relied on manually edited documents to expose REST contracts to clients.
- With SOAP-based Web services, a WSDL serves as a contract for the client and provides a detailed description of the operations and associated request/response payloads.
- The WADL or Web Application Description Language, specification tried to fill this gap in the REST Web services world, but it didn’t get a lot of adoption.
- In recent years, there has been a growth in the number of metadata standards such as Swagger, Apiary, and iODocs for describing REST services.
- Most of them grew out of the need to document APIs, thereby expanding an API’s adoption.

# Documenting REST Services using Swagger

- **Dependency**

```
<dependency>
    <groupId>com.mangofactory</groupId>
    <artifactId>swagger-springmvc</artifactId>
    <version>1.0.2</version>
</dependency>
```

- To integrate Swagger UI in our application, we download the stable version of Swagger UI from the project's GitHub site at: https://github.com/swagger-api/swagger-ui

# Documenting REST Services using Swagger

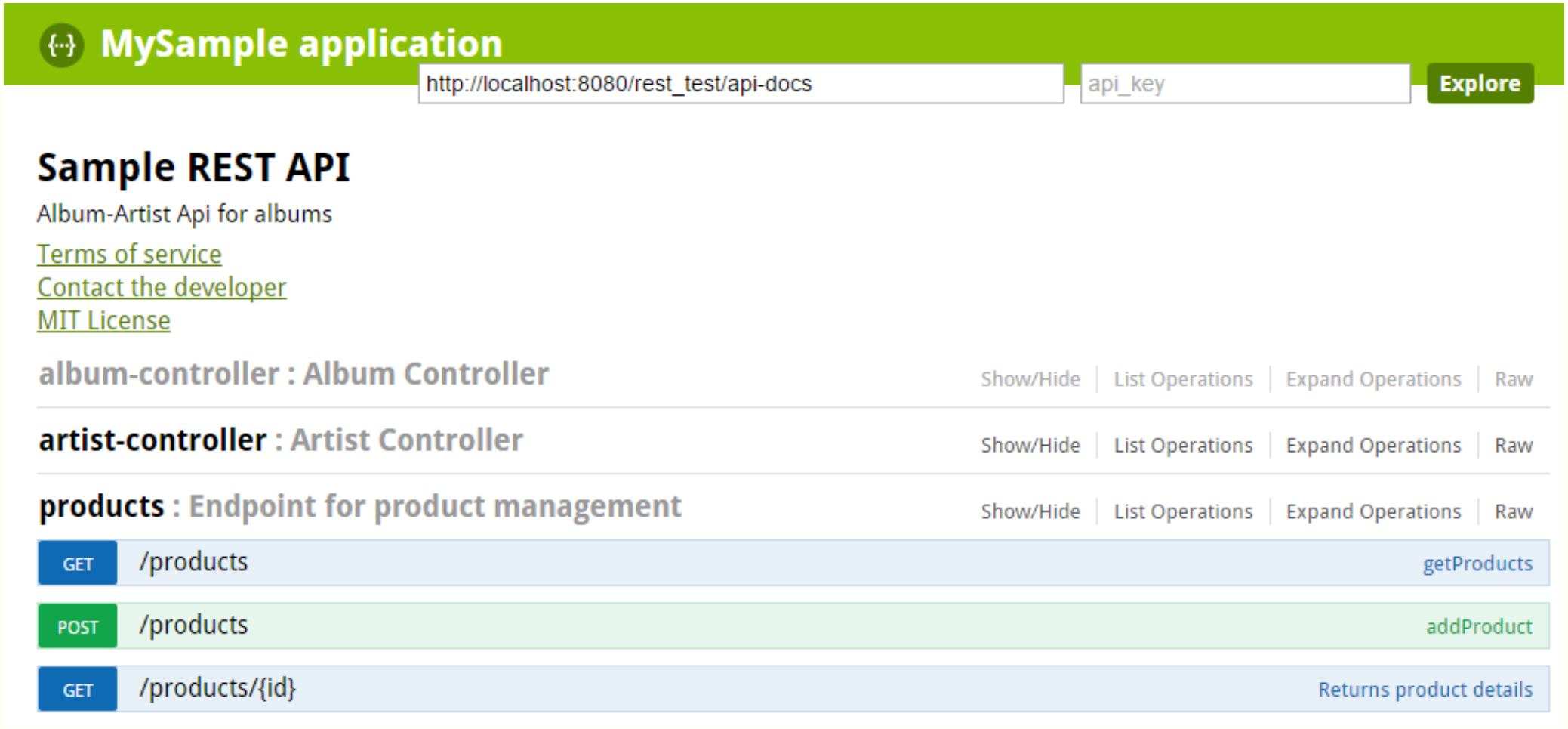
- Modify index.html file as per your requirement

```
    $(function () {
      window.swaggerUi = new SwaggerUi({
      url: "http://localhost:8080/api-docs",
      dom_id: "swagger-ui-container",
       // code removed for brevity
}
```

# Custom Swagger implementation

```
@Configuration
@EnableSwagger
public class SwaggerConfig {
    @Inject
    private SpringSwaggerConfig springSwaggerConfig;

    @Bean
    public SwaggerSpringMvcPlugin configureSwagger() {
        SwaggerSpringMvcPlugin swaggerSpringMvcPlugin = new SwaggerSpringMvcPlugin(
                this.springSwaggerConfig);
        ApiInfo apiInfo = new ApiInfoBuilder().title("Sample REST API")
                .description("Album-Artist Api for albums")
                .termsOfServiceUrl("http://banuprakash.com/terms-of-service")
                .contact("banuprakashc@yahoo.co.in").license("MIT License")
                .licenseUrl("http://opensource.org/licenses/MIT").build();
        swaggerSpringMvcPlugin.apiInfo(apiInfo).apiVersion("1.0");
        return swaggerSpringMvcPlugin;
    }
}
```

# Swagger UI


MySample application
http://localhost:8080/rest_test/api-docs
api_key
Explore
Sample REST API
Album-Artist Api for albums
Terms of service
Contact the developer
MIT License
album-controller : Album Controller
Show/Hide
List Operations
Expand Operations
Raw
artist-controller : Artist Controller
Show/Hide
List Operations
Expand Operations
Raw
products : Endpoint for product management
Show/Hide
List Operations
Expand Operations
Raw
GET
/products
getProducts
POST
/products
addProduct
GET
/products/{id}
Returns product details

# Documenting REST Services using Swagger

- **Creating endpoint documentation @Api**

```
@Api(value = "products", description = "Endpoint for product management")
@Controller
public class ProductController {
```

- **Operations documentation @ApiOperation, @ApiResponse, @ApiParam**

```
@ApiOperation(value = "Returns product details", notes = "Returns a product", response = Product.class)
@ApiResponses(value = {
    @ApiResponse(code = 200, message = "Successful retrieval of product detail", response = Product.class),
    @ApiResponse(code = 404, message = "product with given id does not exist"),
    @ApiResponse(code = 500, message = "Internal server error")}
)
@RequestMapping(value = "/products/{id}", method = RequestMethod.GET)
public @ResponseBody ResponseEntity<Product> getProduct(@ApiParam(name = "id", value = "id of product", required = true)
        @PathVariable("id") int id) {
    System.out.println("Path : " + id);
    return new ResponseEntity<>(productService.getProduct(id),
            HttpStatus.OK);
}
```

# Documenting REST Services using Swagger

- **Creating model documentation @ApiModel, @ApiModelProperty**

```
@XmlRootElement(name = "product")
@XmlAccessorType(XmlAccessType.FIELD)
@ApiModel
public class Product {
    private int id;
    private String name;
    private double price;

    /**
    public Product() {

     * @param id
    public Product(int id, String name, double price) {

    @ApiModelProperty(position = 1, required = true, value = "product id - positive number")
    public int getId() {
        return id;
    }

    @ApiModelProperty(position = 2, required = true, value = "product name alphanumeric")
    public String getName() {
        return name;
    }
```

# Swagger UI

GET /products/{id} Returns product details

## Implementation Notes

Returns a product

## Response Class

Model | Model Schema

**Product {**
  **id** (integer): product id - positive number,
  **name** (string): product name alphanumeric,
  **price** (number, *optional*)
**}**

# Swagger UI

## Parameters

| Parameter | Value | Description | Parameter Type | Data Type |
|---|---|---|---|---|
| id | 1 | **id of product** | path | integer |

## Response Messages

| HTTP Status Code | Reason | Response Model |
|---|---|---|
| 200 | Successful retrieval of product detail | Model \| Model Schema<br>**Product {**<br>**id** (integer): product id - positive number,<br>**name** (string): product name alphanumeric,<br>**price** (number, *optional*)<br>**}** |
| 401 | Unauthorized | |
| 403 | Forbidden | |
| 404 | product with given id does not exist | |
| 500 | Internal server error | |

Try it out! Hide Response

# Spring Security

- In order to use Spring Security you must add the necessary dependencies

```
<dependencies>
  <!-- ... other dependency elements ... -->
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-web</artifactId>
    <version>3.2.8.RELEASE</version>
  </dependency>
  <dependency>
    <groupId>org.springframework.security</groupId>
    <artifactId>spring-security-config</artifactId>
    <version>3.2.8.RELEASE</version>
  </dependency>
</dependencies>
```

# Creating your Spring Security configuration

- The name of the configureGlobal method is not important.
- However, it is important to only configure AuthenticationManagerBuilder in a class annotated with either @EnableWebSecurity, @EnableWebMvcSecurity, @EnableGlobalMethodSecurity, or @EnableGlobalAuthentication.
- Doing otherwise has unpredictable results.

```
@Configuration
@EnableWebSecurity
public class SecurityJavaConfig extends WebSecurityConfigurerAdapter {


    @Override
    public void configure(AuthenticationManagerBuilder auth) throws Exception {
        auth.inMemoryAuthentication().
          withUser("banu").password("prakash").roles("ADMIN").and().
          withUser("rahul").password("prakash").roles("USER");
    }

}
```

# Registering Spring Security

- The MessageSecurityWebApplicationInitializer will automatically register the spring SecurityFilterChain
- Filter for every URL in your application.
- If Filters are added within other WebApplicationInitializer instances we can use @Order to control the ordering of the Filter instances.

```
public class MessageSecurityWebApplicationInitializer
      extends AbstractSecurityWebApplicationInitializer {
}
```

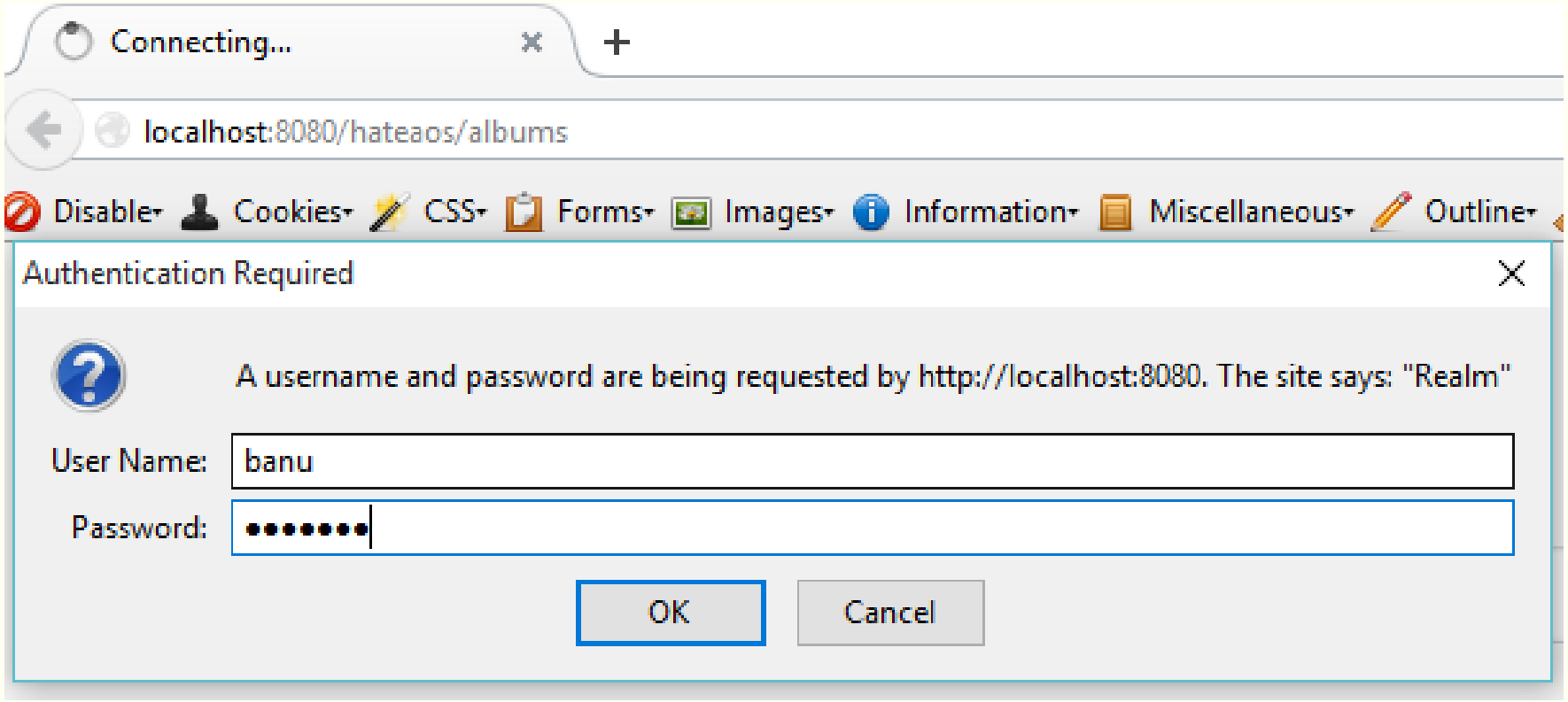
# Loading SecurityConfig

- Application context is initialized using MessageWebApplicationInitializer
- The @ComponentScan is loading all configuration within the specified package (and child packages) as RootConfiguration.

```java
public class MessageWebApplicationInitializer extends
        AbstractAnnotationConfigDispatcherServletInitializer {

    @Override
    protected Class<?>[] getRootConfigClasses() {
        return new Class[] { RootConfiguration.class };
    }

    @Override
    protected Class<?>[] getServletConfigClasses() {
        return new Class[] { WebConfig.class };
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }
}
```

```java
@Configuration
@ComponentScan("com.sample")
public class RootConfiguration {

}
```

# Authenticating to the secured application



# Creating a Custom Login Form (WEB-INF/view/login.jsp)

```
<html xmlns:th="http://www.thymeleaf.org"
    xmlns:tiles="http://www.thymeleaf.org">
<head>
</head>
<body>
    <div tiles:fragment="content">
        <form name="f" th:action="@{/login}" method="post">
            <fieldset>
                <legend>Please Login</legend>
                <label for="username">Username</label>
                <input type="text"
                    id="username" name="username" />
                <label for="password">Password</label>
                <input type="password" id="password" name="password" />
                <div class="form-actions">
                    <button type="submit" class="btn">Log in</button>
                </div>
            </fieldset>
        </form>
    </div>
</body>
</html>
```

# Creating a Custom Login Form

1. The URL we submit our username and password to is the same URL as our login form (i.e. **/login**), but a **POST** instead of a **GET**.
2. When authentication fails, the browser is redirected to **/login?error** so we can display an error message by detecting if the parameter **error** is non-null.
3. When we are successfully logged out, the browser is redirected to **/login?logout** so we can display an logout success message by detecting if the parameter **logout** is non-null.
4. The username should be present on the HTTP parameter username
5. The password should be present on the HTTP parameter password

# Configuring Login Form

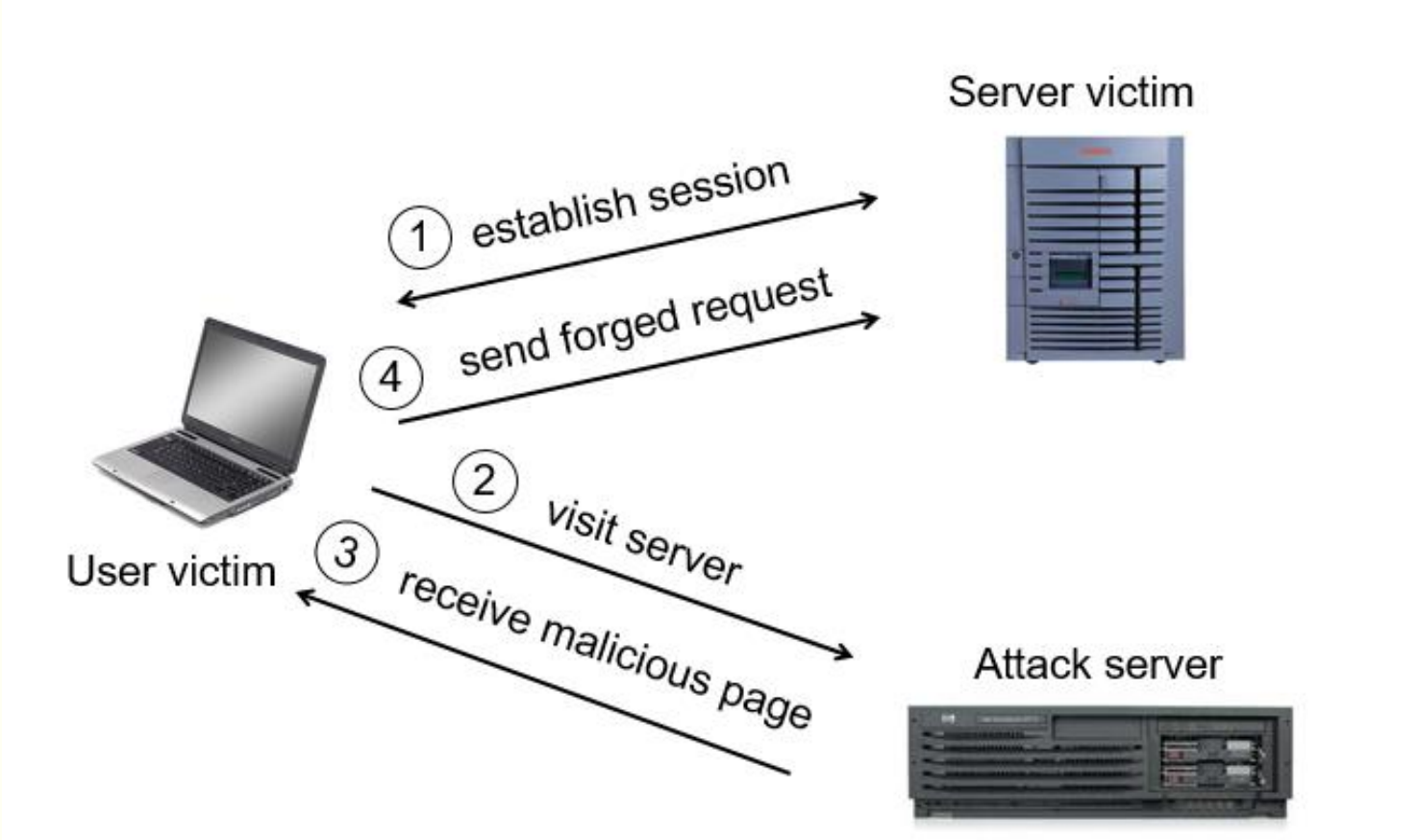
```
@Configuration
@EnableWebSecurity
public class SecurityJavaConfig extends WebSecurityConfigurerAdapter {


    public void configure(AuthenticationManagerBuilder auth) throws Exception {...

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
        .csrf().disable()
        .authorizeRequests()
            .antMatchers("/resources/**").permitAll()
            .antMatchers("/login").permitAll()
            .anyRequest().authenticated()
            .and()
        .formLogin()
            .loginPage("/login")
            .permitAll()
            .and()
        .logout()
            .permitAll();
    }
}
```

# Configure ViewResolver

```
@EnableWebMvc
@Configuration
@ComponentScan({ "com.sample" })
public class WebConfig extends WebMvcConfigurerAdapter {

    public void configureContentNegotiation(..

    public void configureMessageConverters(..

    private HttpMessageConverter<Object> createXmlHttpMessageConverter() {..

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        registry.addViewController("/login").setViewName("login");
        registry.setOrder(Ordered.HIGHEST_PRECEDENCE);
    }

    @Bean
       public ViewResolver viewResolver() {
          InternalResourceViewResolver bean = new InternalResourceViewResolver();

          bean.setViewClass(JstlView.class);
          bean.setPrefix("/WEB-INF/view/");
          bean.setSuffix(".jsp");

          return bean;
       }
}
```

# Cross Site Request Forgery (CSRF) attacks.

- Spring Security has added protection against CSRF attacks



# CSRF and XSS

- CSRF (Cross-site request forgery)
  - When a malicious website causes a user's browser to perform unwanted actions on a trusted website
  - Examples: Transfer money out of user's account, harvest user ids, compromise user accounts
- XSS (Cross-site scripting)
  - Malicious website leverages bugs in trusted website to cause unwanted action on user's browser (circumventing the same-origin policy)
  - Examples: Reading cookies, authentication information, code injection

# Configure CSRF Protection

- CSRF protection is enabled by default with Java configuration
- **Include the CSRF Token**
- The last step is to ensure that you include the CSRF token in all PATCH, POST, PUT, and DELETE methods

```java
@EnableWebSecurity
public class WebSecurityConfig extends
WebSecurityConfigurerAdapter {

@Override
protected void configure(HttpSecurity http) throws Exception {
        http
        .csrf().disable();
}
}
```

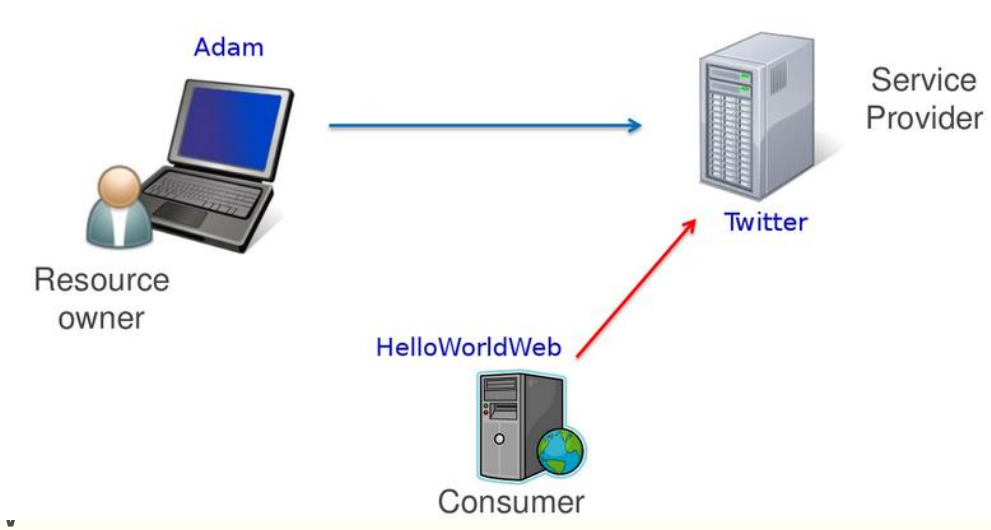
```html
<c:url var="logoutUrl" value="/logout"/>
<form action="${logoutUrl}"
        method="post">
<input type="submit"
        value="Log out" />
<input type="hidden"
        name="${_csrf.parameterName}"
        value="${_csrf.token}"/>
</form>
```

# Authentication using client id

- http://www.facebook.com/dialog/oauth/?client_id=1600137293549207&redirect_uri=https://apps.facebook.com/banu_canvas/

# OAuth

- OAuth is a specification that defines secure authenticatio model on behalf of another user.
- The first party represents a user, in our case Adam, who i called in the OAuth terminology a *Resource Owner*. Adan has an account on Twitter.
- Twitter represents the second party. This party is calle a *Service Provider*.
- Twitter offers a web interface that Adam uses to create ne tweets, read tweets of others etc. Now, Adam uses our ne web site, HelloWorldWeb, that displays the last tweet of the logged in user.
- To do so, web site needs to have access to the Twitter account of Adam. HelloWorldWeb site is a 3rd party application that wants to connect to Twitter and get Adam's tweets. In OAuth, such party is called Consumer.

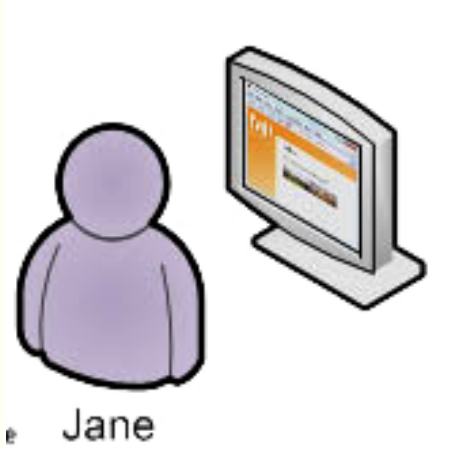


# OAuth

- Two versions of OAuth exists at the moment
  - *OAuth 1* defined by OAuth 1.0 specification
  - *OAuth 2* defined by OAuth 2.0 specification.

# OAuth example

- *Jane wants to share some of her vacation photos with her friends.*
- *Jane uses Faji, a photo sharing site, for sharing journey photos.*
- *She signs into her faji.com account, and uploads two photos which she marks private*
- In OAuth terminology, Jane is the resource owner and Faji the server. The 2 photos Jane uploaded are the protected resources



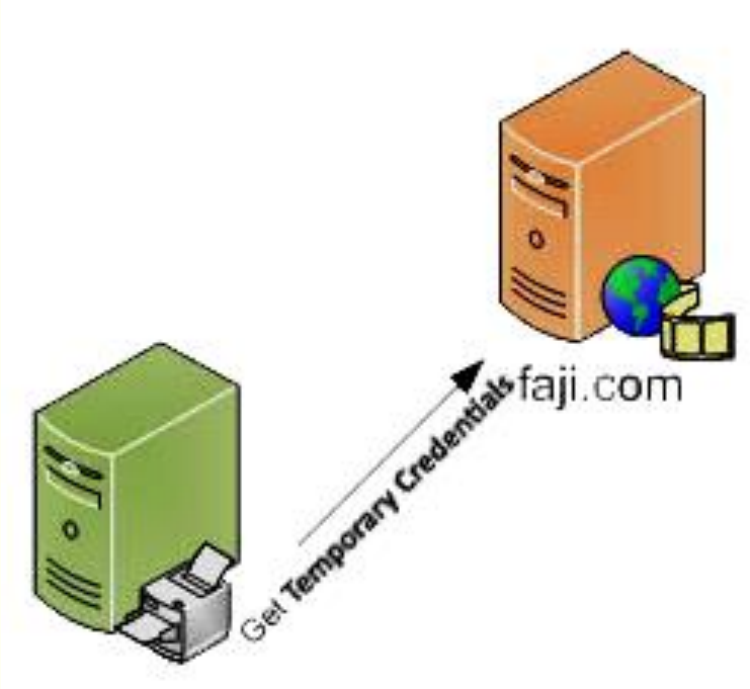

# OAuth example

- *Jane visits beppa.com and begins to order prints. Beppa supports importing images from many photo sharing sites, including Faji. Jane selects the photos source and clicks Continue.*


- In OAuth terminology, Beppa is the client. When Beppa added support for Faji photo import, a Beppa developer known in OAuth as a client developer obtained a set of client credentials (client identifier and secret) from Faji to be used with Faji's OAuth-enabled API.

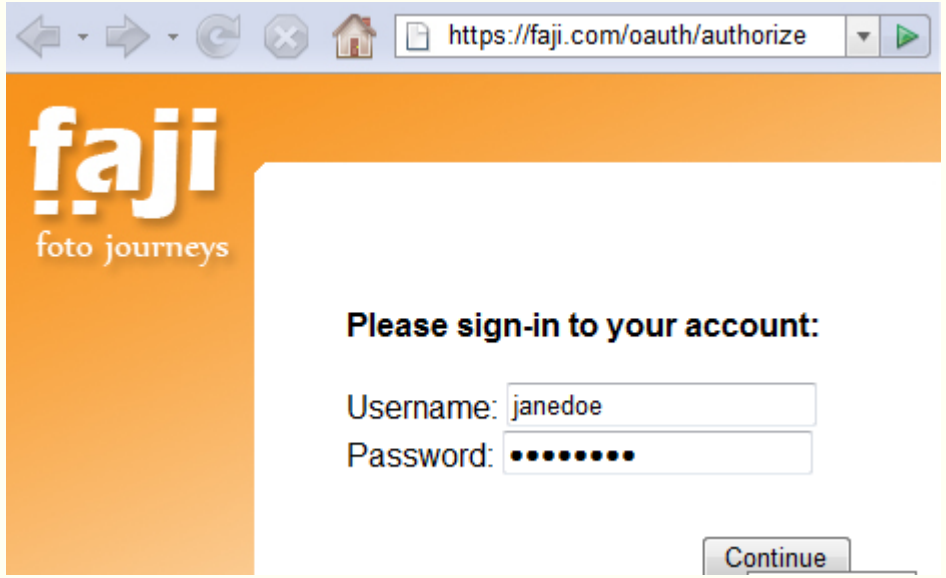
# OAuth example

- After Jane clicks Continue, Beppa requests from Faji a set of temporary credentials.
- At this point, the temporary credentials are not resource-owner-specific, and can be used by Beppa to gain resource owner approval from Jane to access her photos.



# OAuth example

- When Beppa receives the temporary credentials, it redirects Jane to the Faji OAuth User Authorization URL with the temporary credentials and asks Faji to redirect Jane back once approval has been granted to http://beppa.com/order.
- Jane has been redirected to Faji and is requested to sign into the site. OAuth requires that servers first authenticate the resource owner, and then ask them to grant access to the client.

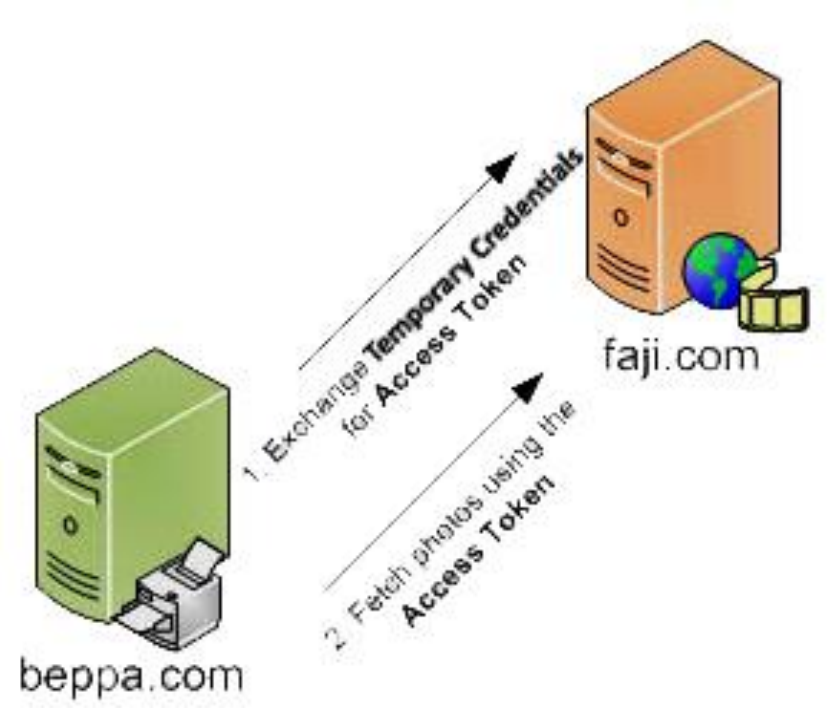


# OAuth example

- After successfully logging into Faji, Jane is asked to grant access to Beppa, the client. Faji informs Jane of who is requesting access (in this case Beppa) and the type of access being granted. Jane can approve or deny access.

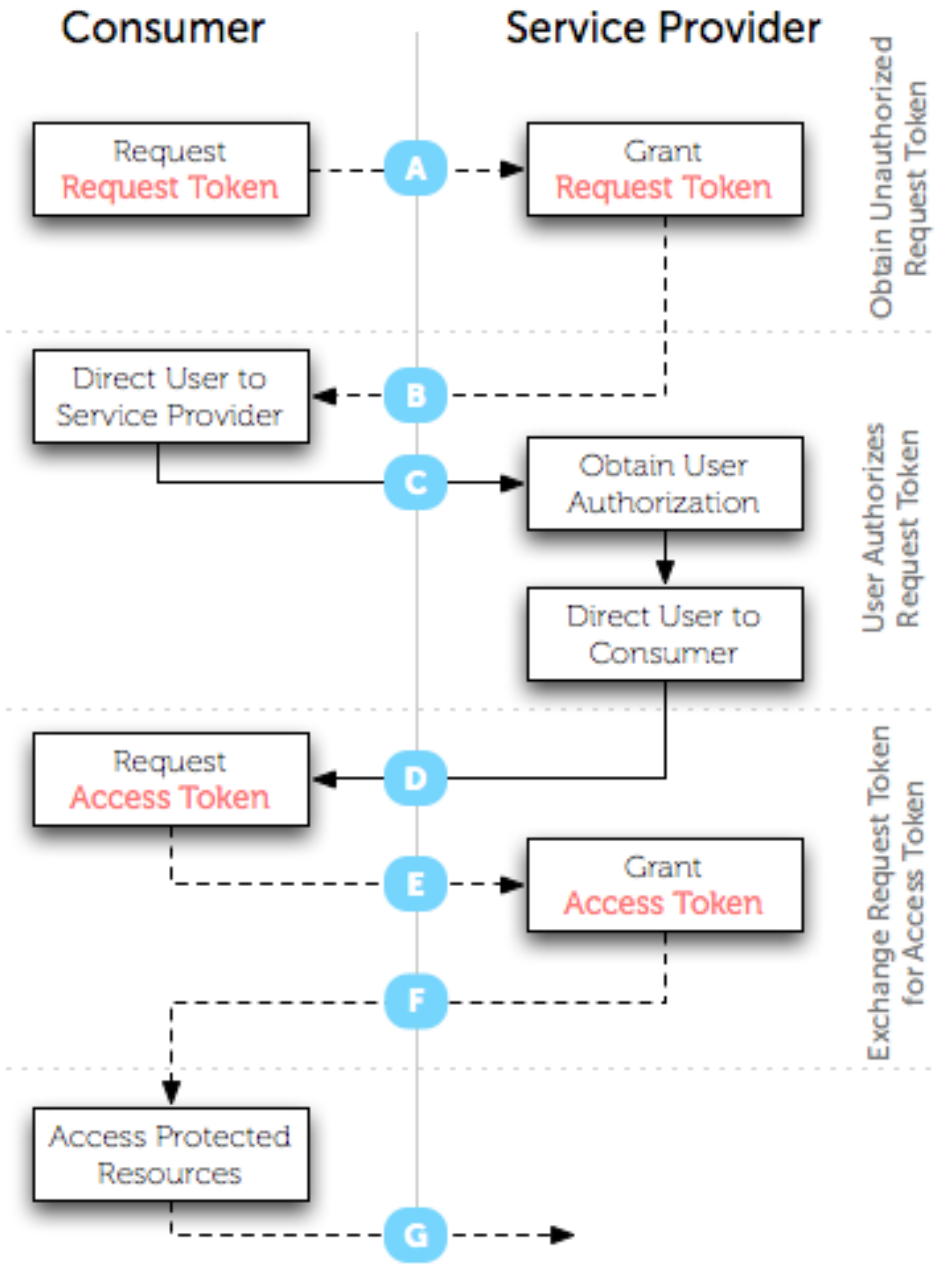
# OAuth example

- Beppa uses the authorized Request Token and exchanges it for an Access Token. Request Tokens are only good for obtaining User approval, while Access Tokens are used to access Protected Resources, in this case Jane's photos. In the first request, Beppa exchanges the Request Token for an Access Token and in the second request gets the photos.



# OAUTH AUTHENTICATION FLOW v1.0a

Person Using Web Browser / Manual Entry
Consumer / Service Provider




**A Consumer Requests Request Token**

Request includes
oauth_consumer_key
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_callback

**B Service Provider Grants Request Token**

Response includes
oauth_token
oauth_token_secret
oauth_callback_confirmed

**C Consumer Directs User to Service Provider**

Request includes
oauth_token (optional)

**D Service Provider Directs User to Consumer**

Request includes
oauth_token
oauth_verifier

**E Consumer Requests Access Token**

Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)
oauth_verifier

**F Service Provider Grants Access Token**

Response includes
oauth_token
oauth_token_secret

**G Consumer Accesses Protected Resources**

Request includes
oauth_consumer_key
oauth_token
oauth_signature_method
oauth_signature
oauth_timestamp
oauth_nonce
oauth_version (optional)

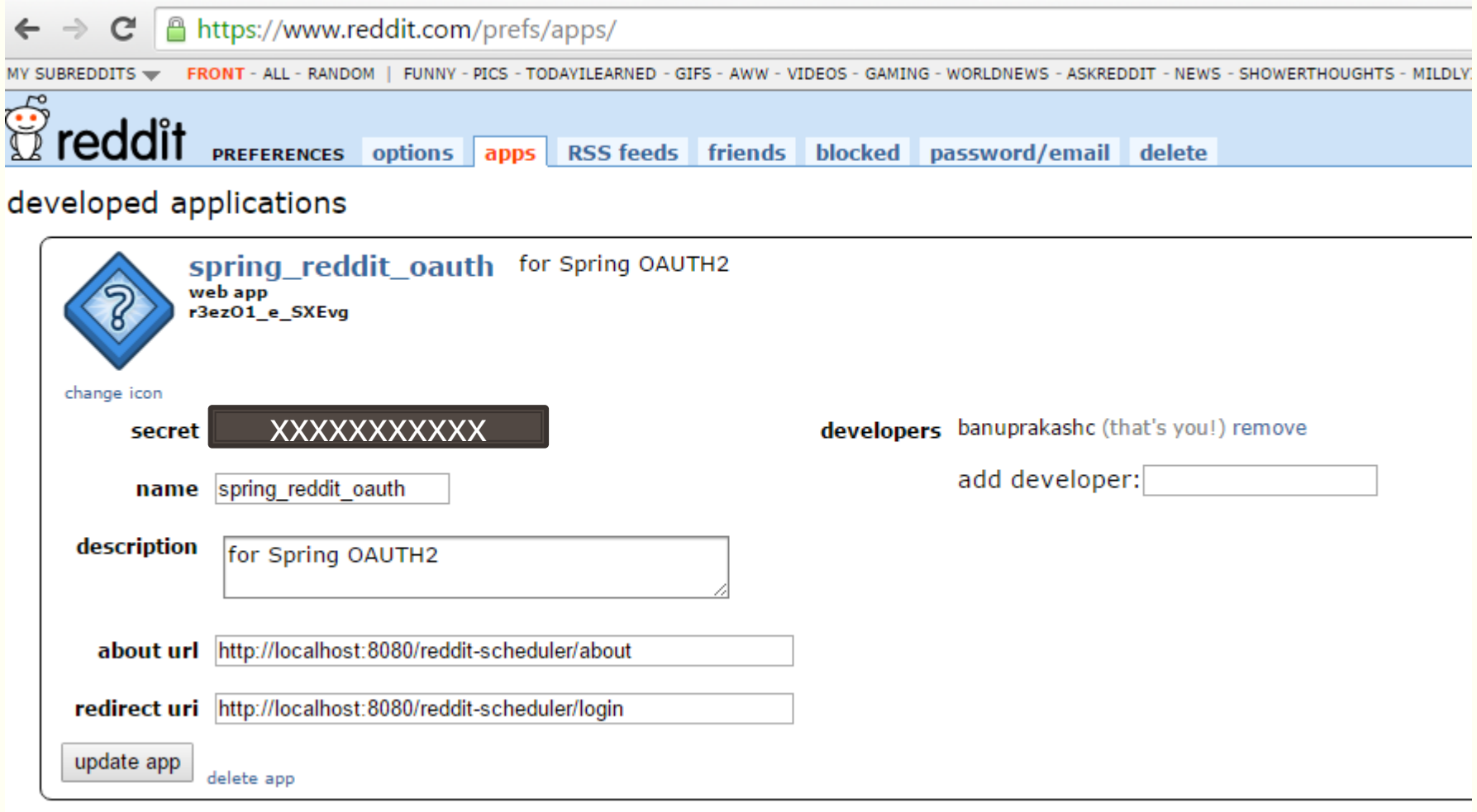
# OAuth Service providers

| Service provider | Oauth V |
|---|---|
| Dropbox | 1 |
| Evernote | 1 |
| Facebook | 2.0 |
| Flickr | 1.0a |
| Foursquare | 2 |
| GitHub | 2 |
| Google | 2 |
| Google App Engine | 1.0a |
| Instagram | 2 |
| LinkedIn | 1.0a, 2.0 |
| Microsoft (Hotmail, Windows Live, Messenger, Xbox) | 2 |

| Service provider | OAuth V |
|---|---|
| MySpace | 1.0a |
| OpenTable | 1.0a |
| PayPal | 2 |
| Tumblr | 1.0a |
| Twitter | 1.0a |
| Ubuntu One | 1 |
| Vimeo | 1.0a |
| Xero | 1.0a |
| XING | 1.0 |
| Yahoo! | 1.0a |
| Yammer | 2 |
| CloudFoundry | 2 |

# OAuth2 and Spring Security

- **Authenticating with Reddit**



# Maven Configuration

- Spring Security OAuth – we need to add the following dependency to our *pom.xml*

```
<dependency>
    <groupId>org.springframework.security.oauth</groupId>
    <artifactId>spring-security-oauth2</artifactId>
    <version>2.0.6.RELEASE</version>
</dependency>
```

# *reddit.properties*

```
clientID=xxxxxxxx
clientSecret=xxxxxxxx
accessTokenUri=https://www.reddit.com/api/v1/access_token
userAuthorizationUri=https://www.reddit.com/api/v1/authorize
```

# Configure OAuth2 Client

```
@Configuration
@EnableOAuth2Client
@PropertySource("classpath:reddit.properties")
public class ResourceConfiguration {

    @Value("${accessTokenUri}")
    private String accessTokenUri;

    @Value("${userAuthorizationUri}")
    private String userAuthorizationUri;

    @Value("${clientID}")
    private String clientID;

    @Value("${clientSecret}")
    private String clientSecret;

    @Bean
    public OAuth2ProtectedResourceDetails reddit() {
        final AuthorizationCodeResourceDetails details = new AuthorizationCodeResourceDetails();
        details.setId("reddit");
        details.setClientId(clientID);
        details.setClientSecret(clientSecret);
        details.setAccessTokenUri(accessTokenUri);
        details.setUserAuthorizationUri(userAuthorizationUri);
        details.setTokenName("oauth_token");
        details.setScope(Arrays.asList("identity"));
        details.setPreEstablishedRedirectUri("http://localhost/login");
        details.setUseCurrentUri(false);
        return details;
    }
```

# Configure OAuth2 Client

```
    @Bean
    public OAuth2RestTemplate redditRestTemplate(OAuth2ClientContext clientContext) {
        OAuth2RestTemplate template = new OAuth2RestTemplate(reddit(), clientContext);
        AccessTokenProvider accessTokenProvider = new AccessTokenProviderChain(
          Arrays.<AccessTokenProvider> asList(
            new MyAuthorizationCodeAccessTokenProvider(),
            new ImplicitAccessTokenProvider(),
            new ResourceOwnerPasswordAccessTokenProvider(),
            new ClientCredentialsAccessTokenProvider())
        );
        template.setAccessTokenProvider(accessTokenProvider);
        return template;
    }

}
```

# ServerInitializer

```java
public class ServletInitializer extends AbstractDispatcherServletInitializer {

    @Override
    protected WebApplicationContext createServletApplicationContext() {
        AnnotationConfigWebApplicationContext context =
          new AnnotationConfigWebApplicationContext();
        context.register(WebConfig.class, SecurityConfig.class);
        return context;
    }

    @Override
    protected String[] getServletMappings() {
        return new String[] { "/" };
    }

    @Override
    protected WebApplicationContext createRootApplicationContext() {
        return null;
    }

    @Override
    public void onStartup(ServletContext servletContext) throws ServletException {
        super.onStartup(servletContext);
        registerProxyFilter(servletContext, "oauth2ClientContextFilter");
        registerProxyFilter(servletContext, "springSecurityFilterChain");
    }

    private void registerProxyFilter(ServletContext servletContext, String name) {
        DelegatingFilterProxy filter = new DelegatingFilterProxy(name);
        filter.setContextAttribute(
          "org.springframework.web.servlet.FrameworkServlet.CONTEXT.dispatcher");
        servletContext.addFilter(name, filter).addMappingForUrlPatterns(null, false, "/*");
    }
}
```

# MVC Configuration

```
@Configuration
@EnableWebMvc
@ComponentScan(basePackages = { "org.baeldung.web" })
public class WebConfig extends WebMvcConfigurerAdapter {

    @Bean
    public static PropertySourcesPlaceholderConfigurer
      propertySourcesPlaceholderConfigurer() {
        return new PropertySourcesPlaceholderConfigurer();
    }

    @Bean
    public ViewResolver viewResolver() {
        InternalResourceViewResolver viewResolver = new InternalResourceViewResolver();
        viewResolver.setPrefix("/WEB-INF/jsp/");
        viewResolver.setSuffix(".jsp");
        return viewResolver;
    }

    @Override
    public void configureDefaultServletHandling(
      DefaultServletHandlerConfigurer configurer) {
        configurer.enable();
    }

    public void addResourceHandlers(ResourceHandlerRegistry registry) {
        registry.addResourceHandler("/resources/**").addResourceLocations("/resources/");
    }

    @Override
    public void addViewControllers(ViewControllerRegistry registry) {
        super.addViewControllers(registry);
        registry.addViewController("/home.html");
    }
}
```

# Security Configuration

```
@Configuration
@EnableWebSecurity
public class SecurityConfig extends WebSecurityConfigurerAdapter {

    @Override
    protected void configure(AuthenticationManagerBuilder auth)
      throws Exception {
        auth.inMemoryAuthentication();
    }

    @Override
    protected void configure(HttpSecurity http) throws Exception {
        http
            .anonymous().disable()
            .csrf().disable()
            .authorizeRequests()
            .antMatchers("/home.html").hasRole("USER")
            .and()
            .httpBasic()
            .authenticationEntryPoint(oauth2AuthenticationEntryPoint());
    }

    private LoginUrlAuthenticationEntryPoint oauth2AuthenticationEntryPoint() {
        return new LoginUrlAuthenticationEntryPoint("/login");
    }
}
```

# *RedditController*

- We use method *redditLogin()* to get the user information from his Reddit account and load an authentication from it

```
@Controller
public class RedditController {

    @Autowired
    private OAuth2RestTemplate redditRestTemplate;

    @RequestMapping("/login")
    public String redditLogin() {
        JsonNode node = redditRestTemplate.getForObject(
          "https://oauth.reddit.com/api/v1/me", JsonNode.class);
        UsernamePasswordAuthenticationToken auth =
          new UsernamePasswordAuthenticationToken(node.get("name").asText(),
          redditRestTemplate.getAccessToken().getValue(),
          Arrays.asList(new SimpleGrantedAuthority("ROLE_USER")));

        SecurityContextHolder.getContext().setAuthentication(auth);
        return "redirect:home.html";
    }

}
```

# *home.jsp*

```
<%@ taglib prefix="c" uri="http://java.sun.com/jsp/jstl/core"%>
<%@ taglib prefix="sec" uri="http://www.springframework.org/security/tags"%>
<html>
<body>
    <h1>Welcome, <small><sec:authentication property="principal.username" /></small></h1>
</body>
</html>
```