vmware | SPRING & CLOUD FOUNDRY 开发者大会

中国北京 - 2012年12月7日-8日

Multi Client Development with Spring

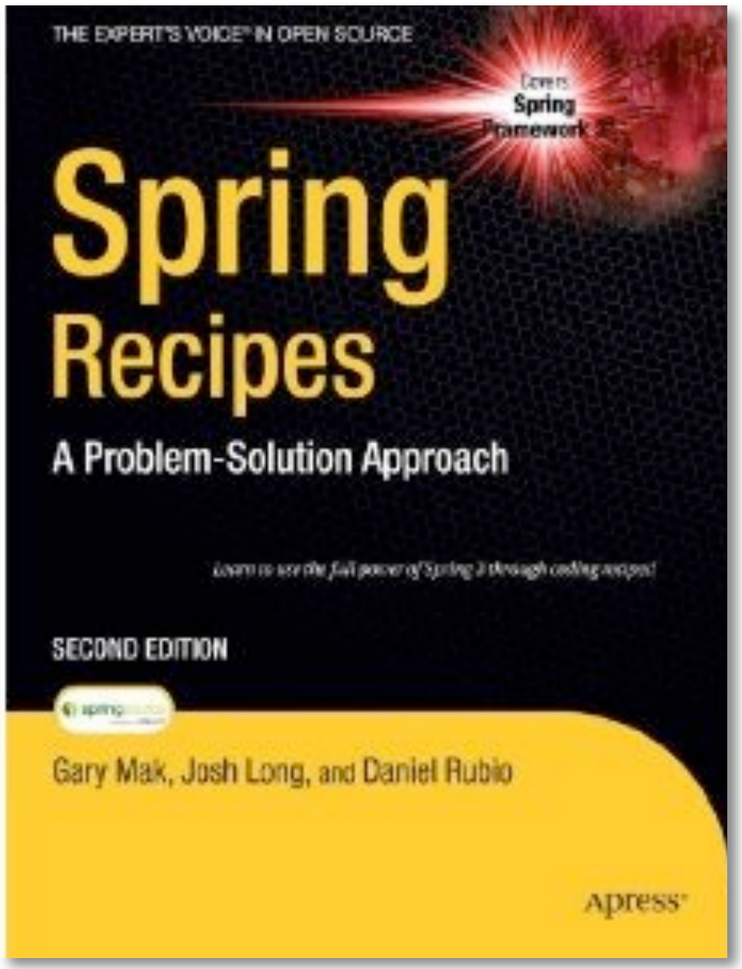
Josh Long

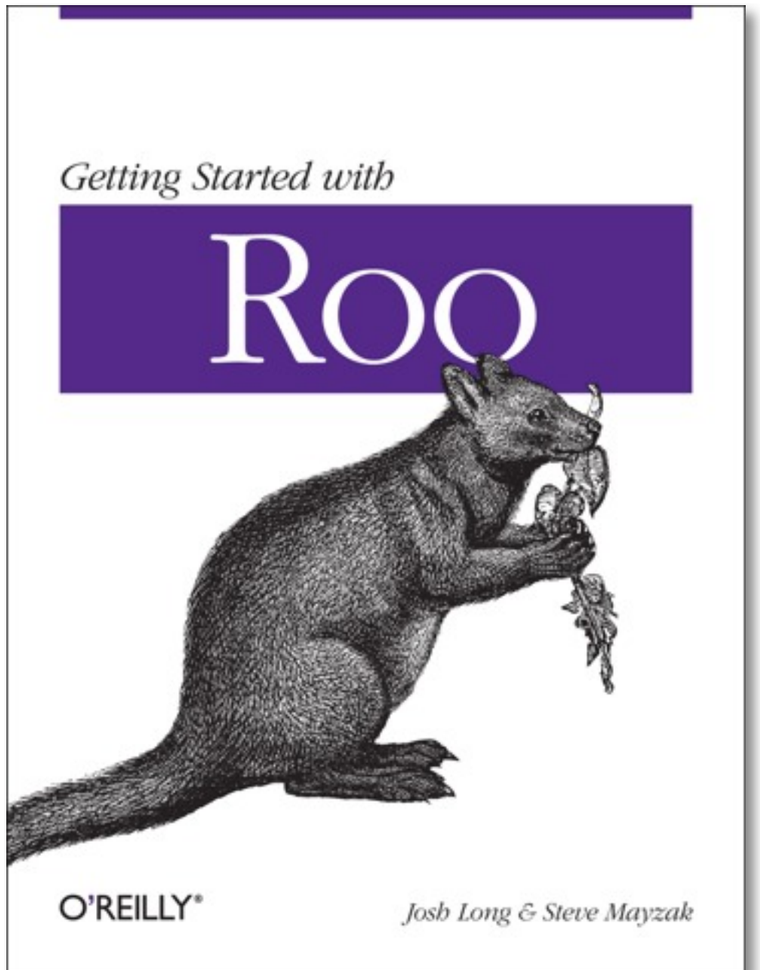
Spring Developer Advocate, *SpringSource, a Division of VMWare*

http://www.joshlong.com || @starbuxman || josh.long@springsource.com

About Josh Long (龙之春)

Spring Developer Advocate
twitter: @starbuxman
weibo: @springsource
josh.long@springsource.com



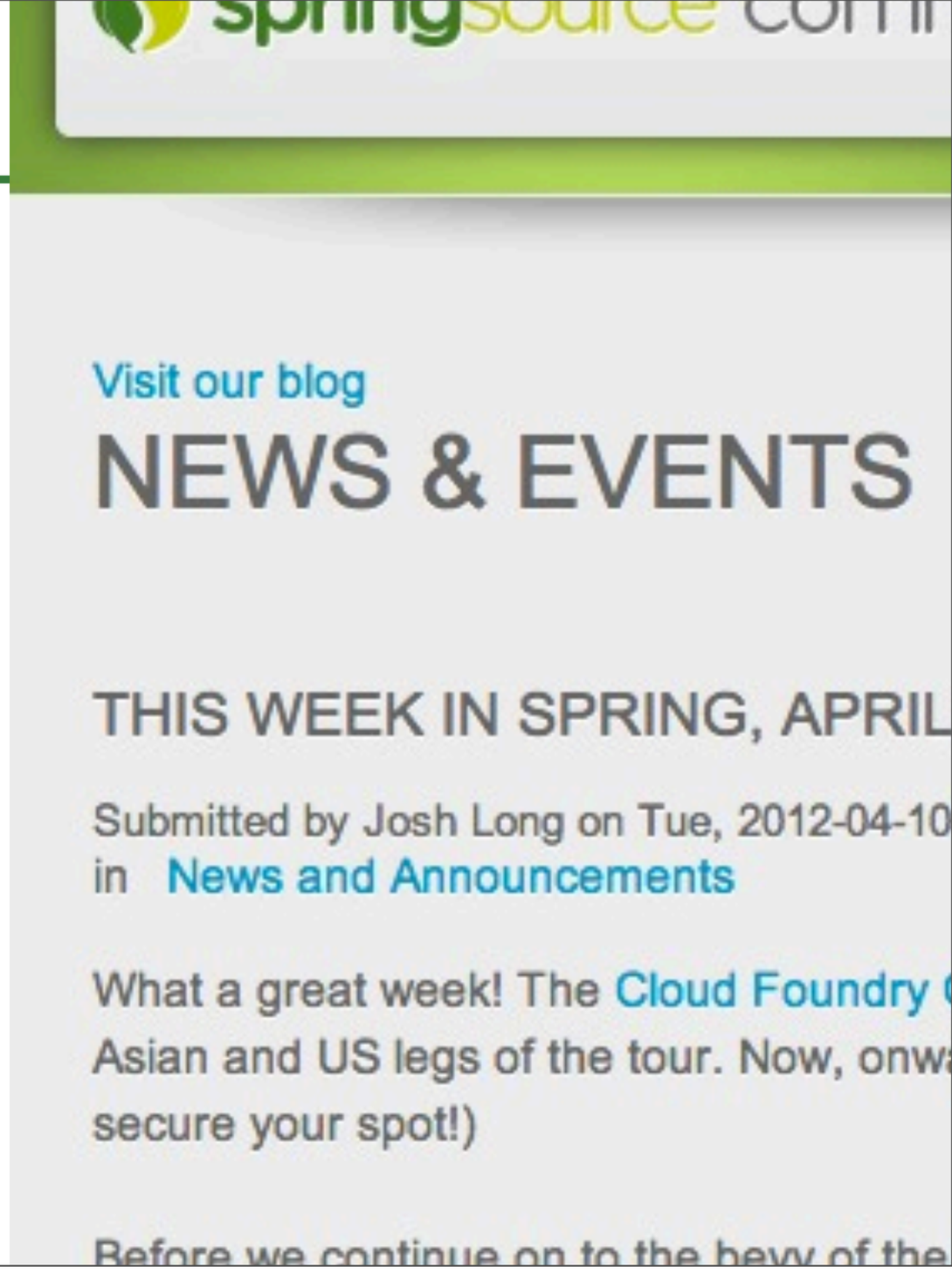


About Josh Long

Spring Developer Advocate
twitter: @starbuxman
josh.long@springsource.com

Contributor To:

- Spring Integration
- Spring Batch
- Spring Hadoop
- Activiti Workflow Engine
- Akka Actor engine



Why Are We Here?

> Software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

-Bob Martin

Why Are We Here?

do NOT reinvent the Wheel!

Spring's aim:

bring simplicity to java development

web tier & RIA

service tier

batch processing

integration & messaging

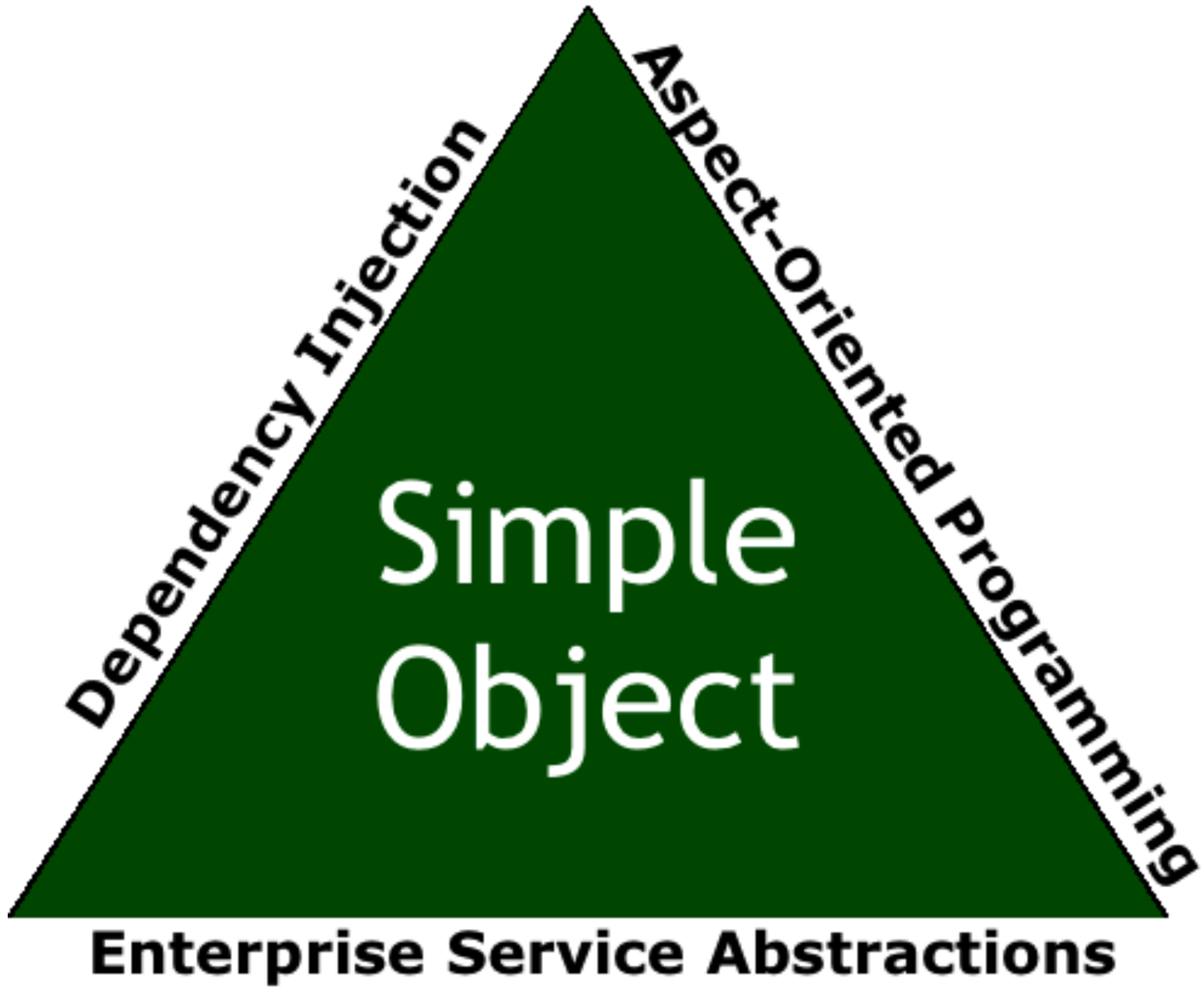
data access / NoSQL / Big Data

mobile

The Spring framework

the cloud:	lightweight	traditional
CloudFoundry Google App Engine Amazon BeanStalk Others	tc Server Tomcat Jetty	WebSphere JBoss AS WebLogic (on legacy versions, too!)

Not All Sides Are Equal



Spring Web Support

New in 3.1: XML-free web applications

- In a servlet 3 container, you can also use Java configuration, negating the need for **`web.xml`**:

```
public class SampleWebApplicationInitializer implements WebApplicationInitializer {

   public void onStartup(ServletContext sc) throws ServletException {

        AnnotationConfigWebApplicationContext ac =
                    new AnnotationConfigWebApplicationContext();
        ac.setServletContext(sc);
        ac.scan( “a.package.full.of.services”, “a.package.full.of.controllers” );

        sc.addServlet("spring", new DispatcherServlet(webContext));

   }
}
```

Thin, Thick, Web, Mobile and Rich Clients: Web Core

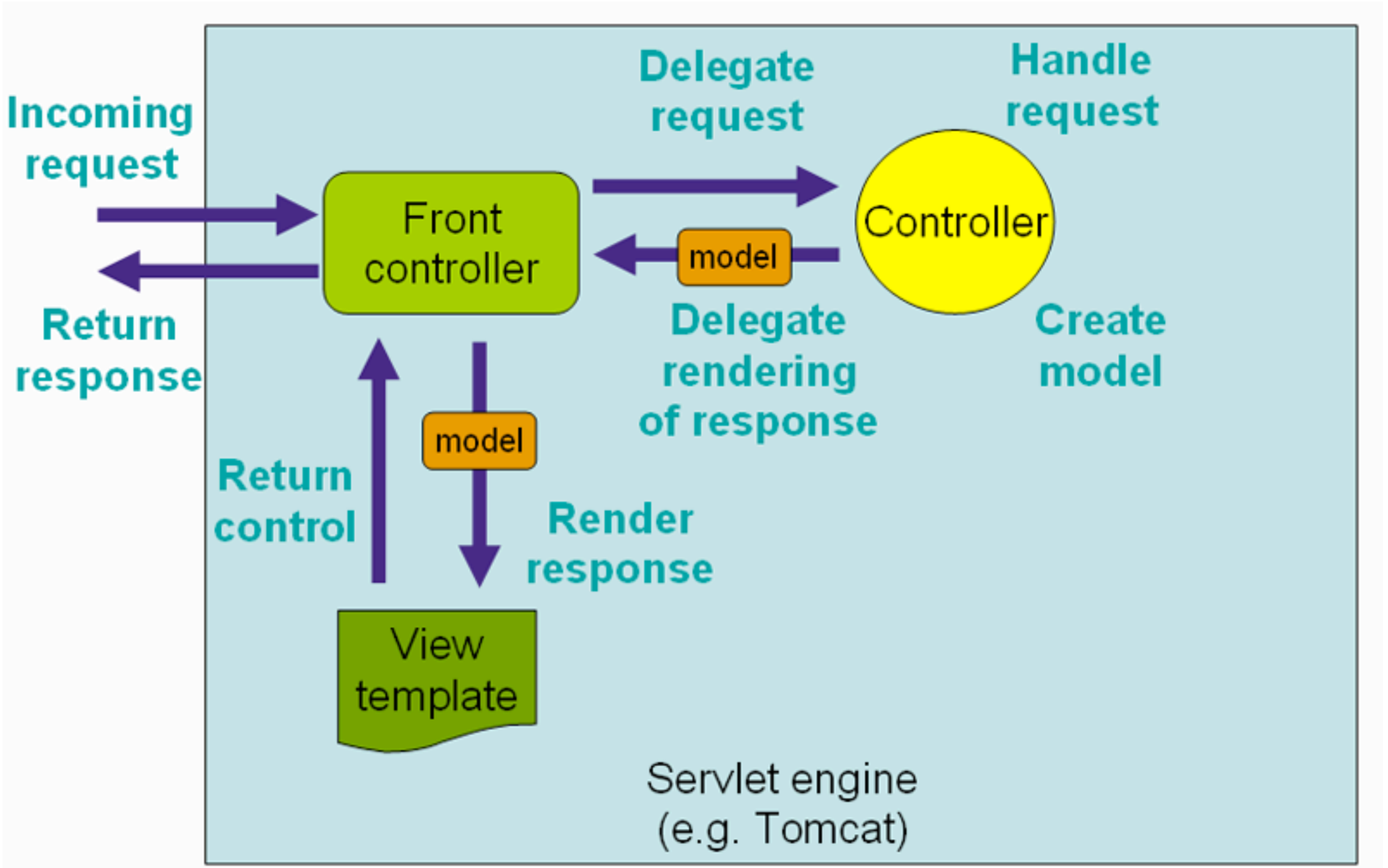
Spring Dispatcher Servlet

- Objects don't *have* to be web-specific.
- Spring web supports lower-level web machinery: ‘
  - **HttpRequestHandler** (supports remoting: Caucho, Resin, JAX RPC)
  - **DelegatingFilterProxy**.
  - **HandlerInterceptor** wraps requests to **HttpRequestHandlers**
  - **ServletWrappingController** lets you force requests to a servlet through the Spring Handler chain
  - **OncePerRequestFilter** ensures that an action only occurs once, no matter how many filters are applied. Provides a nice way to avoid duplicate filters
- Spring provides access to the Spring application context using **WebApplicationContextUtils**, which has a static method to look up the context, even in environments where Spring isn't managing the web components

Thin, Thick, Web, Mobile and Rich Clients: Web Core

- **Spring provides the easiest way to integrate with your web framework of choice**
  - Spring Faces for JSF 1 and 2
  - Struts support for Struts 1
  - Tapestry, Struts 2, Stripes, Wicket, Vaadin, Play framework, etc.
  - GWT, Flex

Thin, Thick, Web, Mobile and Rich Clients: Spring MVC



The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {


}
```

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value="/url/of/my/resource")
    public String processTheRequest() {
        // ...
        return "home";
    }

}
```

GET http://127.0.0.1:8080/**url/of/my/resource**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value="/url/of/my/resource",
                    method = RequestMethod.GET)
    public String processTheRequest() {
        // ...
        return "home";
    }

}
```

GET http://127.0.0.1:8080/**url/of/my/resource**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value="/url/of/my/resource",
                    method = RequestMethod.GET)
    public String processTheRequest( HttpServletRequest request) {
        String contextPath = request.getContextPath();
        // ...
        return "home";
    }

}
```

GET http://127.0.0.1:8080/**url/of/my/resource**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/resource”,
                    method = RequestMethod.GET)
    public String processTheRequest( @RequestParam(“search”) String searchQuery ) {
        // ...
        return “home”;
    }

}
```

GET http://127.0.0.1:8080/**url/of/my/resource?search=Spring**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/{id}”,
                    method = RequestMethod.GET)
    public String processTheRequest( @PathVariable(“id”) Long id) {
        // ...
        return “home”;
    }

}
```

GET http://127.0.0.1:8080/**url/of/my/2322**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value="/url/of/my/{id}" )
    public String processTheRequest( @PathVariable("id") Long customerId,
                                     Model model ) {
        model.addAttribute("customer", service.getCustomerById( customerId ) );
        return "home";
    }

    @Autowired CustomerService service;

}
```

GET http://127.0.0.1:8080/**url/of/my/232**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/resource”,
                    method = RequestMethod.GET)
    public String processTheRequest( HttpServletRequest request, Model model) {
        return “home”;
    }

}
```

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/resource”,
                    method = RequestMethod.GET)
    public View processTheRequest( HttpServletRequest request, Model model) {
        return new XsltView(...);
    }

}
```

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

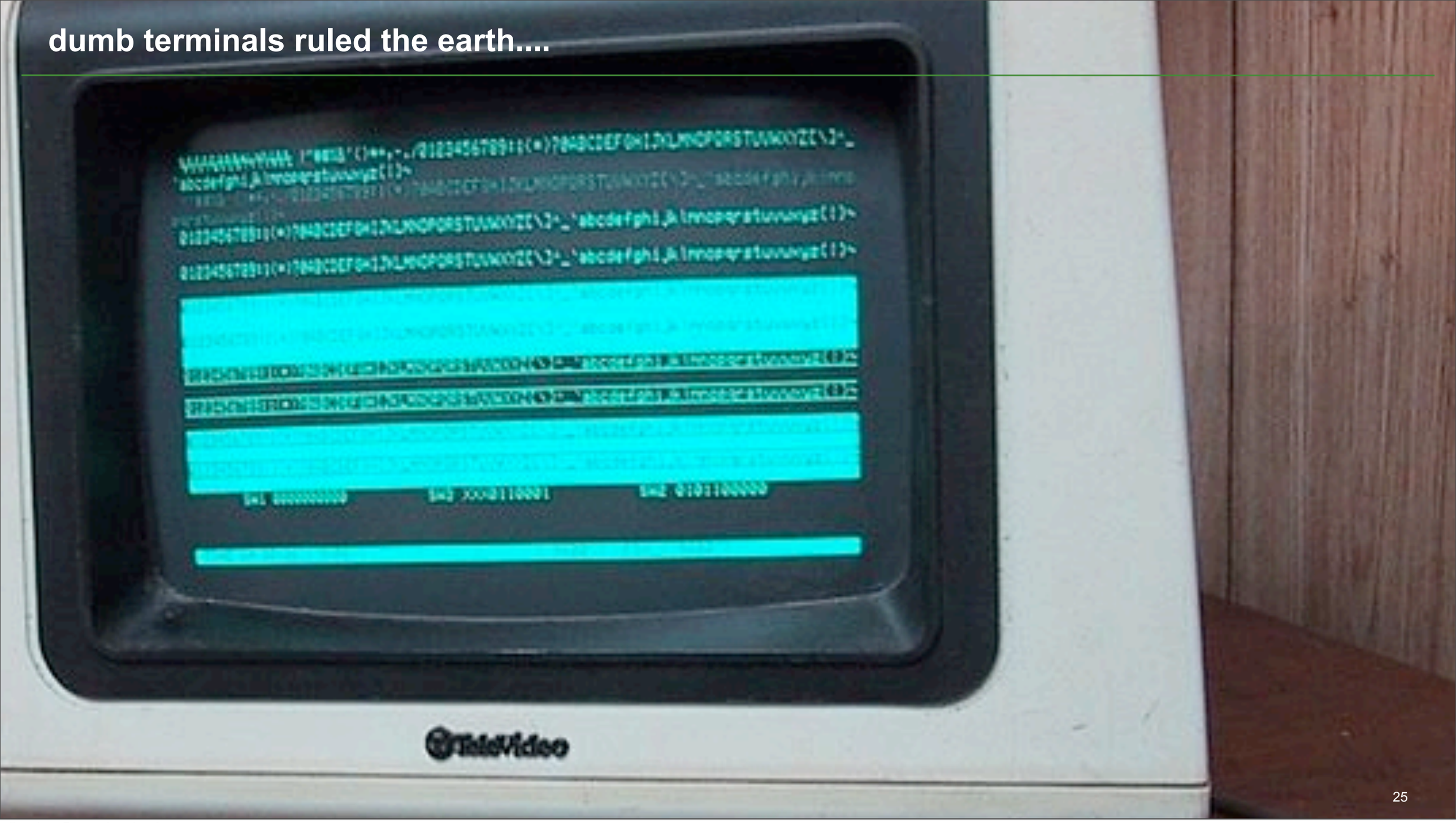
```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/resource”,
                    method = RequestMethod.GET)
    public InputStream processTheRequest( HttpServletRequest request, Model model) {
        return new FileInputStream( ...) ;
    }

}
```

DEMO

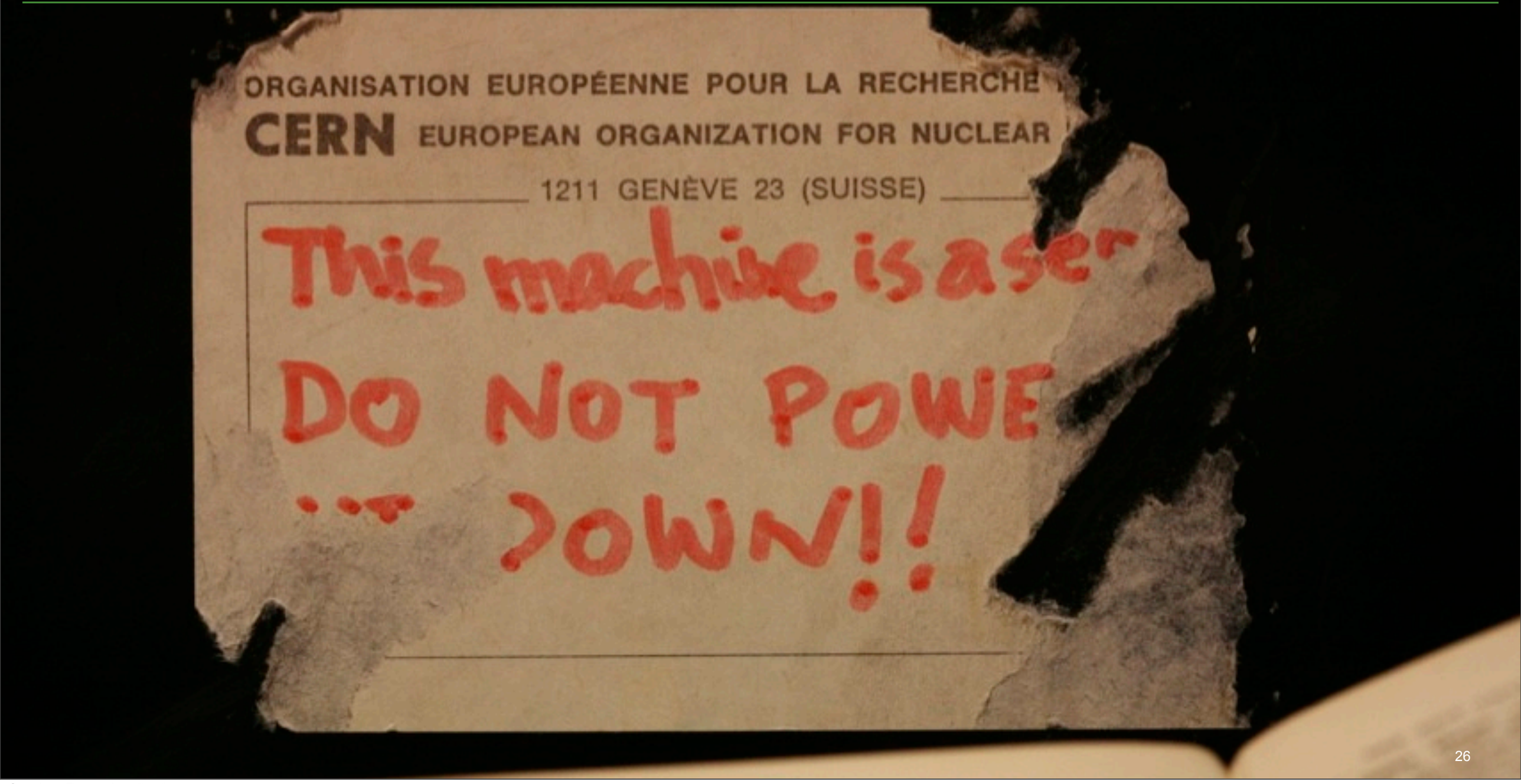
- **Demos**
  - Simple Spring MVC based Application

the new hotness...

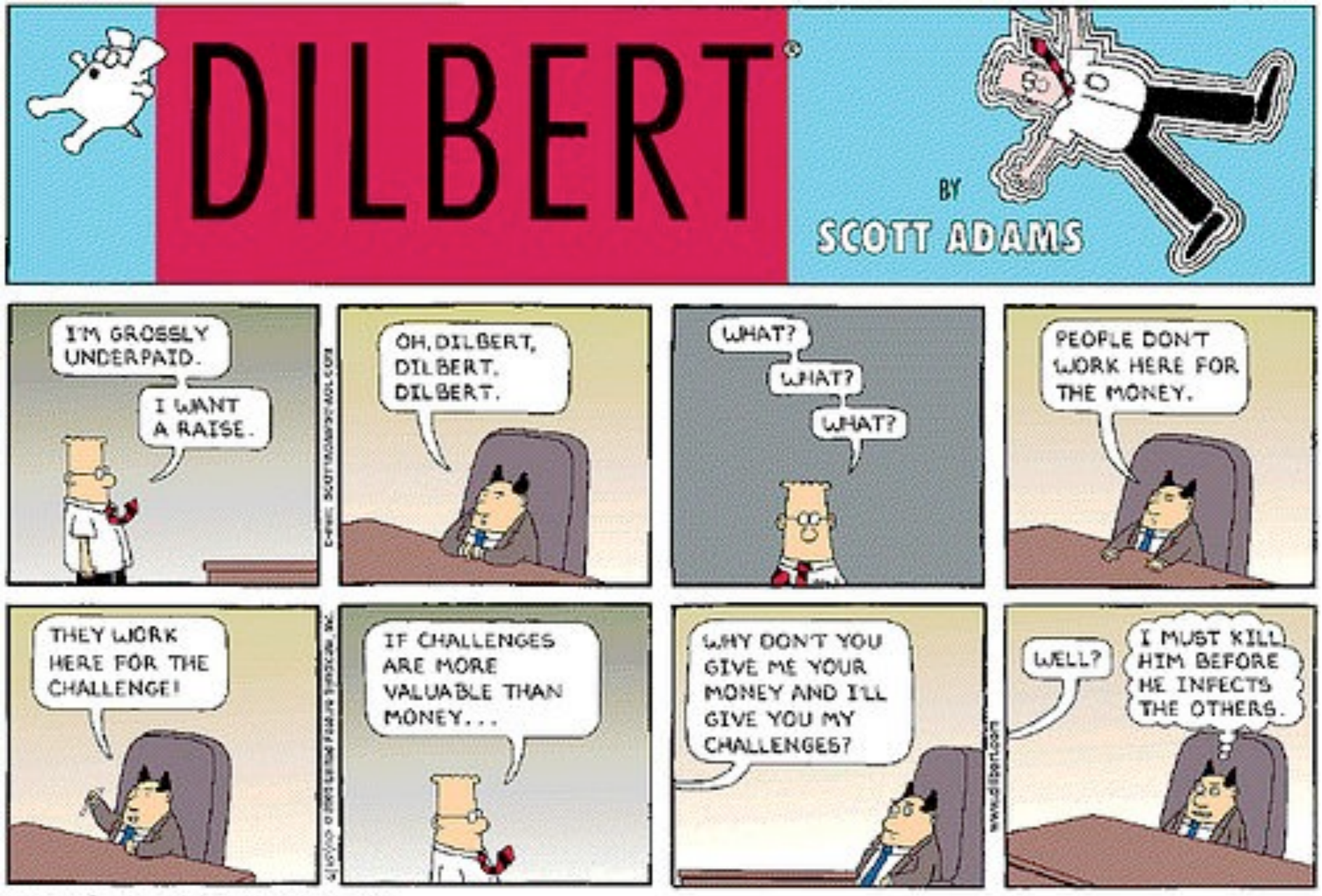
dumb terminals ruled the earth....



then something *magical* happened: the web

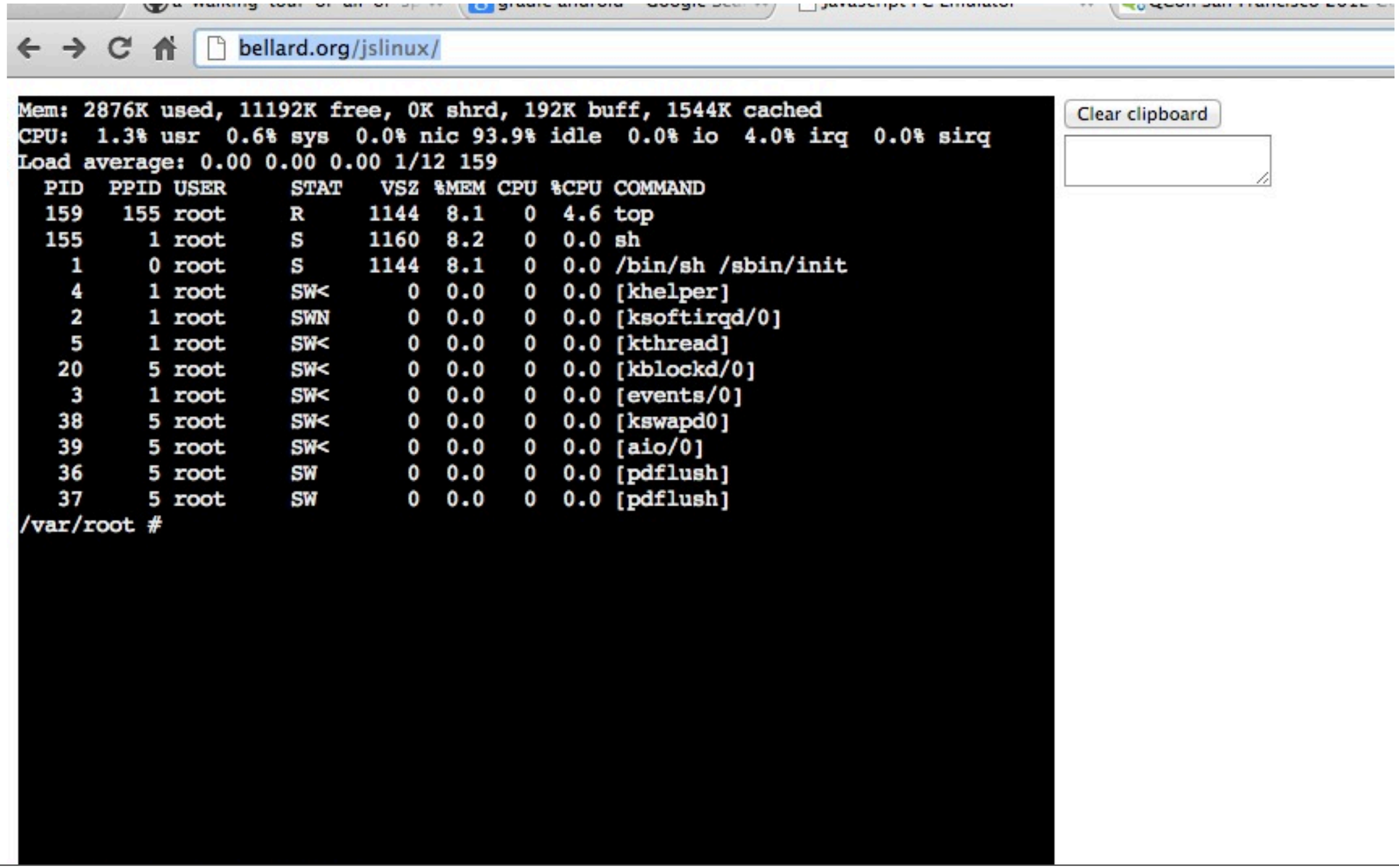


but the web didn't know how to do UI state... so we hacked...



....then the web stopped sucking

How Powerful is JavaScript? ...It Boots Linux!!



REST

Thin, Thick, Web, Mobile and Rich Clients: REST

- **Origin**
  - The term Representational State Transfer was introduced and defined in 2000 by **Roy Fielding** in his doctoral dissertation.
- **His paper suggests these four design principles:**
  - Use HTTP methods explicitly.
    - POST, GET, PUT, DELETE
    - CRUD operations can be mapped to these existing methods
  - Be stateless.
    - State dependencies limit or restrict scalability
  - Expose directory structure-like URIs.
    - URI's should be easily understood
  - Transfer XML, JavaScript Object Notation (JSON), or both.
    - Use XML or JSON to represent data objects or attributes

REST on the Server

- **Spring MVC is basis for REST support**
  - Spring's server side REST support is based on the standard controller model
- **JavaScript and HTML5 can consume JSON-data payloads**

REST on the Client

- **RestTemplate**
  - provides dead simple, idiomatic RESTful services consumption
  - can use Spring OXM, too.
  - Spring Integration and Spring Social both build on the RestTemplate where possible.
- **Spring supports numerous converters out of the box**
  - JAXB
  - JSON (Jackson)

Building clients for your RESTful services: RestTemplate

- **Google search example**

```
RestTemplate restTemplate = new RestTemplate();
String url = "https://ajax.googleapis.com/ajax/services/search/web?v=1.0&q={query}";
String result = restTemplate.getForObject(url, String.class, "SpringSource");
```

- **Multiple parameters**

```
RestTemplate restTemplate = new RestTemplate();
String url = "http://example.com/hotels/{hotel}/bookings/{booking}";
String result = restTemplate.getForObject(url, String.class, "42", “21”);
```

Thin, Thick, Web, Mobile and Rich Clients: `RestTemplate`

- **RestTemplate class is the heart the client-side story**
  - Entry points for the six main HTTP methods
    - DELETE - delete(...)
    - GET - getForObject(...)
    - HEAD - headForHeaders(...)
    - OPTIONS - optionsForAllow(...)
    - POST - postForLocation(...)
    - PUT - put(...)
    - any HTTP operation - exchange(...) and execute(...)

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/resource”,
                    method = RequestMethod.GET)
    public @ResponseBody Customer processTheRequest(  ... ) {
       Customer c = service.getCustomerById( id) ;
       return c;
    }

    @Autowired CustomerService service;

}
```

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value="/url/of/my/someurl",
                    method = RequestMethod.POST)
    public String processTheRequest( @RequestBody Customer postedCustomerObject) {
        // ...
        return "home";
    }

}
```

POST http://127.0.0.1:8080/**url/of/my/someurl**

The Anatomy of a Spring MVC @Controller

Spring MVC configuration – config

```
@Controller
public class CustomerController {

    @RequestMapping(value=”/url/of/my/{id}”,
                    method = RequestMethod.POST)
    public String processTheRequest( @PathVariable(“id”) Long customerId,
                                     @RequestBody Customer postedCustomerObject) {
        // ...
        return “home”;
    }

}
```

POST http://127.0.0.1:8080/**url/of/my/someurl**

What About the Browsers, Man?

- **Problem**: modern browsers only speak **GET** and **POST**
- **Solution:** use Spring's `HiddenHttpMethodFilter` only then send `_method` request parameter in the request

```
<filter>
      <filter-name>hiddenHttpMethodFilter</filter-name>
      <filter-class>org.springframework.web.filter.HiddenHttpMethodFilter</filter-class>
   </filter>
   <filter-mapping>
      <filter-name>hiddenHttpMethodFilter</filter-name>
      <url-pattern>/</url-pattern>
      <servlet-name>appServlet</servlet-name>
   </filter-mapping>
```

DEMO

- **Demos:**
  - Spring REST service
  - Spring REST client

Spring Mobile

Thin, Thick, Web, Mobile and Rich Clients: Mobile

- **Best strategy? Develop Native**
  - Fallback to client-optimized web applications
- **Spring MVC 3.1 mobile client-specific content negotiation and rendering**
  - for *other* devices
    - *(there are other devices besides Android??)*

DEMO

- **Demos:**
  - Mobile clients using client specific rendering

Spring Android

Thin, Thick, Web, Mobile and Rich Clients: Mobile

- **Spring REST is ideal for mobile devices**
- **Spring MVC 3.1 mobile client-specific content negotiation and rendering**
  - for other devices
- **Spring Android**
  - RestTemplate

OK, so.....


Does Spring Android
support dependency injection?
NO.
Y U NOO SUPPORT
DEPENDENCY INJECTION???
CHALLENGE ACCEPTED

Thin, Thick, Web, Mobile and Rich Clients: Mobile

CustomerServiceClient – client

```
private <T> T extractResponse( ResponseEntity<T> response) {
      if (response != null && response().value() == 200) {
           return response.getBody();
      }
      throw new RuntimeException("couldn't extract response.");
}

@Override
public Customer updateCustomer(long id, String fn, String ln) {
      String urlForPath = urlForPath("customer/{customerId}");
      return  extractResponse(this.restTemplate.postForEntity(
            urlForPath, new Customer(id, fn, ln), Customer.class, id));
}
```

Thin, Thick, Web, Mobile and Rich Clients: Mobile

- **Demos:**
  - consuming the Spring REST service from Android

Social Communication

Spring Social

- **Extension to Spring Framework to enable connectivity with Software-as-a-Service providers**
- **Features...**
  - An extensible connection framework
  - A connect controller
  - Java API bindings
  - A sign-in controller
- **http://www.springsource.org/spring-social**

Spring Social Projects

- Spring Social Core
- Spring Social Facebook
- Spring Social Twitter
- Spring Social LinkedIn
- Spring Social TripIt
- Spring Social GitHub
- Spring Social Gowalla
- **Spring Social Weibo** https://github.com/liuce/spring-social-weibo
- **Spring Social Samples**
  - Includes Showcase, Quickstart, Movies, Canvas, Twitter4J, Popup

Spring Social's Key Components

- **Connection Factories**
  - Creates connections; Handles back-end of authorization flow
- **Connect Controller**
  - Orchestrates the web-based connection flow
- **Connection Repository**
  - Persists connections for long-term use
- **Connection Factory Locator**
  - Used by connect controller and connection repository to find connection factories
- **API Bindings**
  - Perform requests to APIs, binding to domain objects, error-handling
- **Provider Sign-In Controller**
  - Signs a user into an application based on an existing connection

Key Steps to Socializing an Application

- **Configure Spring Social beans**
  - Connection Factory Locator and Connection Factories
  - Connection Repository
  - Connect Controller
  - API Bindings
- **Create connection status views**
- **Inject/use API bindings**

Configuration: ConnectionFactoryLocator

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionFactoryLocator connectionFactoryLocator() {
  ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

  registry.addConnectionFactory(
      new TwitterConnectionFactory(
          environment.getProperty("twitter.consumerKey"),
          environment.getProperty("twitter.consumerSecret")));

  registry.addConnectionFactory(
      new FacebookConnectionFactory(
          environment.getProperty("facebook.clientId"),
          environment.getProperty("facebook.clientSecret")));

  return registry;
}
```

Configuration: ConnectionFactoryLocator

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionFactoryLocator connectionFactoryLocator() {
  ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

  registry.addConnectionFactory(
      new TwitterConnectionFactory(
          environment.getProperty("twitter.consumerKey"),
          environment.getProperty("twitter.consumerSecret")));

  registry.addConnectionFactory(
      new FacebookConnectionFactory(
          environment.getProperty("facebook.clientId"),
          environment.getProperty("facebook.clientSecret")));

  return registry;
}
```

Configuration: ConnectionFactoryLocator

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionFactoryLocator connectionFactoryLocator() {
  ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

  registry.addConnectionFactory(
      new TwitterConnectionFactory(
          environment.getProperty("twitter.consumerKey"),
          environment.getProperty("twitter.consumerSecret")));

  registry.addConnectionFactory(
      new FacebookConnectionFactory(
          environment.getProperty("facebook.clientId"),
          environment.getProperty("facebook.clientSecret")));

  return registry;
}
```

Configuration: ConnectionFactoryLocator

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionFactoryLocator connectionFactoryLocator() {
  ConnectionFactoryRegistry registry = new ConnectionFactoryRegistry();

  registry.addConnectionFactory(
      new TwitterConnectionFactory(
          environment.getProperty("twitter.consumerKey"),
          environment.getProperty("twitter.consumerSecret")));

  registry.addConnectionFactory(
      new FacebookConnectionFactory(
          environment.getProperty("facebook.clientId"),
          environment.getProperty("facebook.clientSecret")));

  return registry;
}
```

Configuration: Connection Repository

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionRepository connectionRepository() {
  Authentication authentication = SecurityContextHolder.getContext().
              getAuthentication();
  if (authentication == null) {
    throw new IllegalStateException(
        "Unable to get a ConnectionRepository: no user signed in");
  }
  return usersConnectionRepository().createConnectionRepository(
          authentication.getName());
}
```

Configuration: Connection Repository

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public ConnectionRepository connectionRepository() {
  Authentication authentication = SecurityContextHolder.getContext().
              getAuthentication();
  if (authentication == null) {
    throw new IllegalStateException(
        "Unable to get a ConnectionRepository: no user signed in");
  }
  return usersConnectionRepository().createConnectionRepository(
          authentication.getName());
}
```

```
@Bean
@Scope(value="singleton", proxyMode=ScopedProxyMode.INTERFACES)
public UsersConnectionRepository usersConnectionRepository() {
  return new JdbcUsersConnectionRepository(
              dataSource,
              connectionFactoryLocator(),
              Encryptors.noOpText());
}
```

Configuration: ConnectController

```
@Bean
public ConnectController connectController() {
  return new ConnectController(connectionFactoryLocator(),
                               connectionRepository());
}
```

Configuration: ConnectController

```
@Bean
public ConnectController connectController() {
  return new ConnectController(connectionFactoryLocator(),
                               connectionRepository());
}
```

Configuration: ConnectController

```
@Bean
public ConnectController connectController() {
  return new ConnectController(connectionFactoryLocator(),
                  connectionRepository());
}
```

Configuration: API Bindings

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Facebook facebook() {
  Connection<Facebook> facebook =
          connectionRepository().findPrimaryConnection(Facebook.class);
  return facebook != null ? facebook.getApi() : new FacebookTemplate();
}

@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Twitter twitter() {
  Connection<Twitter> twitter =
          connectionRepository().findPrimaryConnection(Twitter.class);
  return twitter != null ? twitter.getApi() : new TwitterTemplate();
}
```

Configuration: API Bindings

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Facebook facebook() {
  Connection<Facebook> facebook =
          connectionRepository().findPrimaryConnection(Facebook.class);
  return facebook != null ? facebook.getApi() : new FacebookTemplate();
}

@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Twitter twitter() {
  Connection<Twitter> twitter =
          connectionRepository().findPrimaryConnection(Twitter.class);
  return twitter != null ? twitter.getApi() : new TwitterTemplate();
}
```

Configuration: API Bindings

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Facebook facebook() {
  Connection<Facebook> facebook =
        connectionRepository().findPrimaryConnection(Facebook.class);
  return facebook != null ? facebook.getApi() : new FacebookTemplate();
}

@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Twitter twitter() {
  Connection<Twitter> twitter =
        connectionRepository().findPrimaryConnection(Twitter.class);
  return twitter != null ? twitter.getApi() : new TwitterTemplate();
}
```

Configuration: API Bindings

```
@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Facebook facebook() {
  Connection<Facebook> facebook =
          connectionRepository().findPrimaryConnection(Facebook.class);
  return facebook != null ? facebook.getApi() : new FacebookTemplate();
}

@Bean
@Scope(value="request", proxyMode=ScopedProxyMode.INTERFACES)
public Twitter twitter() {
  Connection<Twitter> twitter =
          connectionRepository().findPrimaryConnection(Twitter.class);
  return twitter != null ? twitter.getApi() : new TwitterTemplate();
}
```

Injecting and Using the API Bindings

```
@Controller
public class TwitterTimelineController {

    private final Twitter twitter;

    @Inject
    public TwitterTimelineController(Twitter twitter) {
        this.twitter = twitter;
    }

    @RequestMapping(value="/twitter/tweet",
                    method=RequestMethod.POST)
    public String postTweet(String message) {
        twitter.timelineOperations().updateStatus(message);
        return "redirect:/twitter";
    }
}
```

Injecting and Using the API Bindings

```
@Controller
public class TwitterTimelineController {

    private final Twitter twitter;

    @Inject
    public TwitterTimelineController(Twitter twitter) {
        this.twitter = twitter;
    }

    @RequestMapping(value="/twitter/tweet",
                    method=RequestMethod.POST)
    public String postTweet(String message) {
        twitter.timelineOperations().updateStatus(message);
        return "redirect:/twitter";
    }
}
```

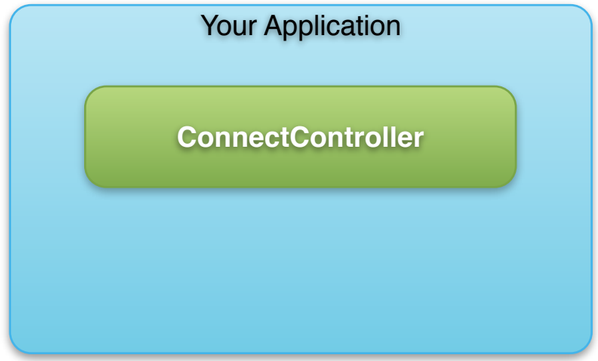
Injecting and Using the API Bindings

```
@Controller
public class TwitterTimelineController {

    private final Twitter twitter;

    @Inject
    public TwitterTimelineController(Twitter twitter) {
        this.twitter = twitter;
    }

    @RequestMapping(value="/twitter/tweet",
                    method=RequestMethod.POST)
    public String postTweet(String message) {
        twitter.timelineOperations().updateStatus(message);
        return "redirect:/twitter";
    }
}
```

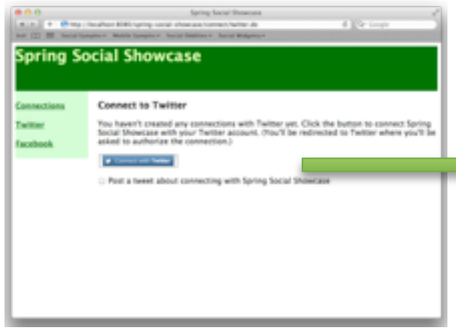
ConnectController Endpoints

- **GET /connect**
  - Displays connection status for all providers
- **GET /connect/{provider}**
  - Displays connection status for a given provider
- **POST /connect/{provider}**
  - Initiates the authorization flow, redirecting to the provider
- **GET /connect/{provider}?oauth_token={token}**
  - Handles an OAuth 1 callback
- **GET /connect/{provider}?code={authorization code}**
  - Handles an OAuth 2 callback
- **DELETE /connect/{provider}**
  - Removes all connections for a user to the given provider
- **DELETE /connect/{provider}/{provider user ID}**
  - Removes a specific connection for the user to the given provider

ConnectController Flow



ConnectController Flow



GET /connect/{provider ID}

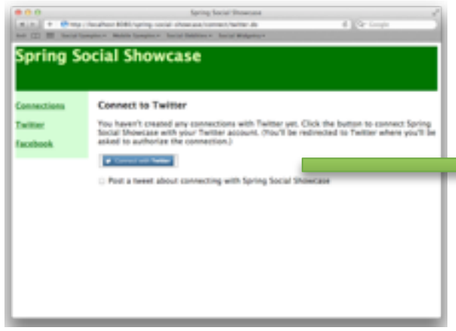
Your Application

ConnectController

Service Provider
(Twitter, Facebook, etc)

Display connection status page

ConnectController Flow



POST /connect/{provider ID}

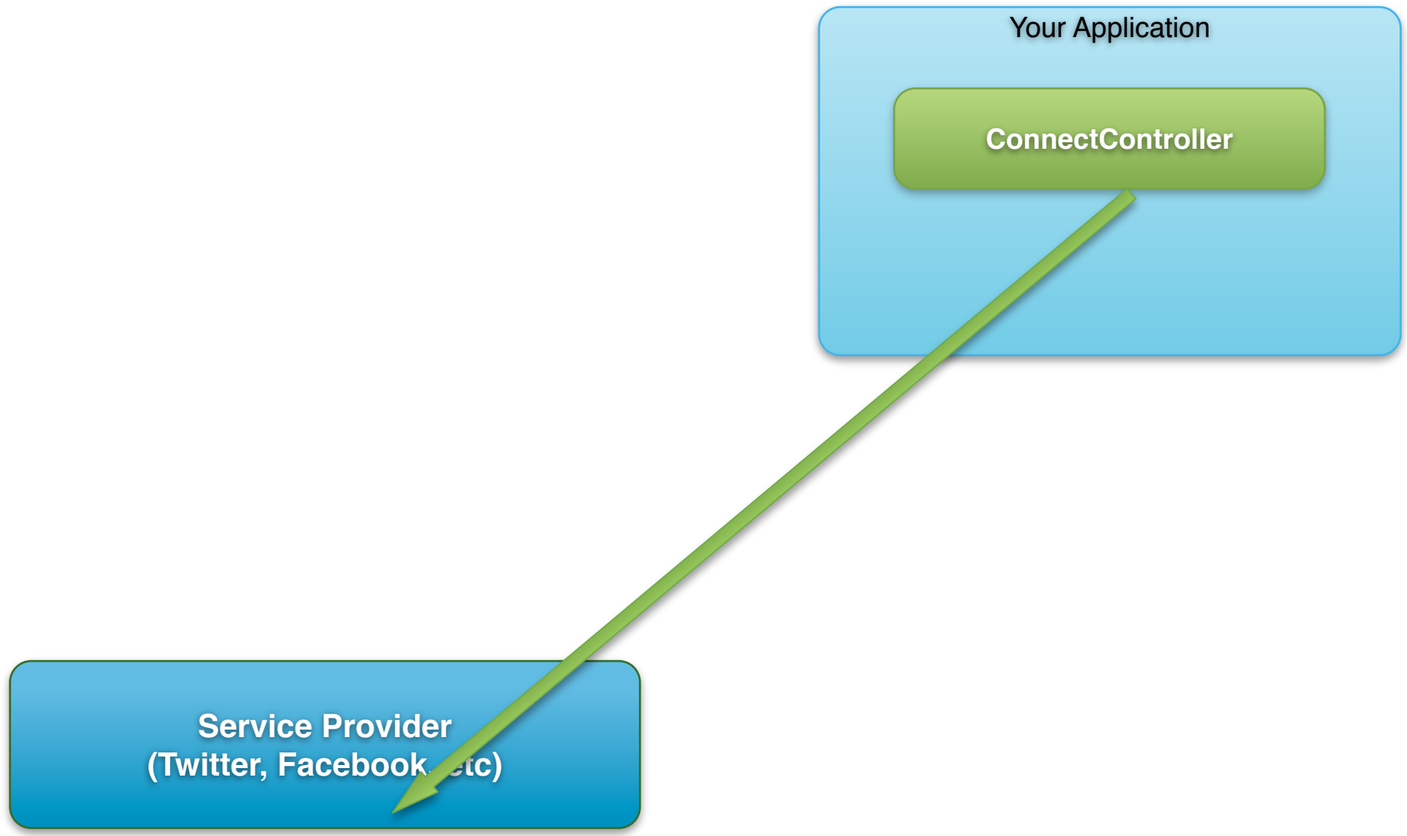
Your Application

ConnectController

Service Provider
(Twitter, Facebook, etc)

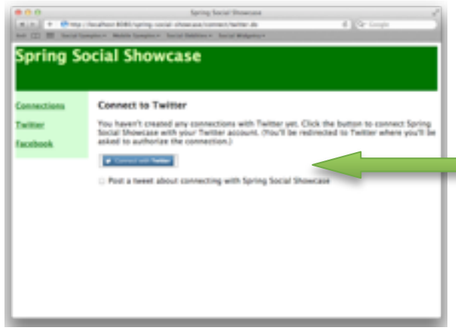
Initiate connection flow

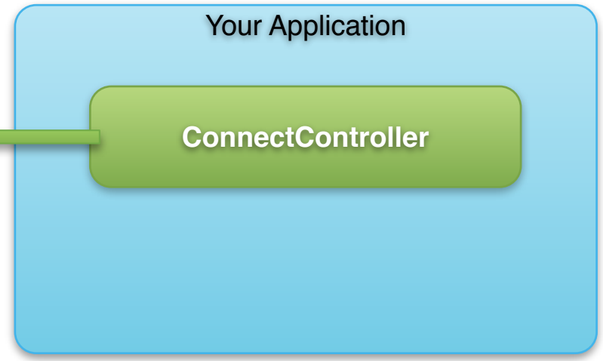
ConnectController Flow



Fetch request token (OAuth 1.0/1.0a only)

ConnectController Flow

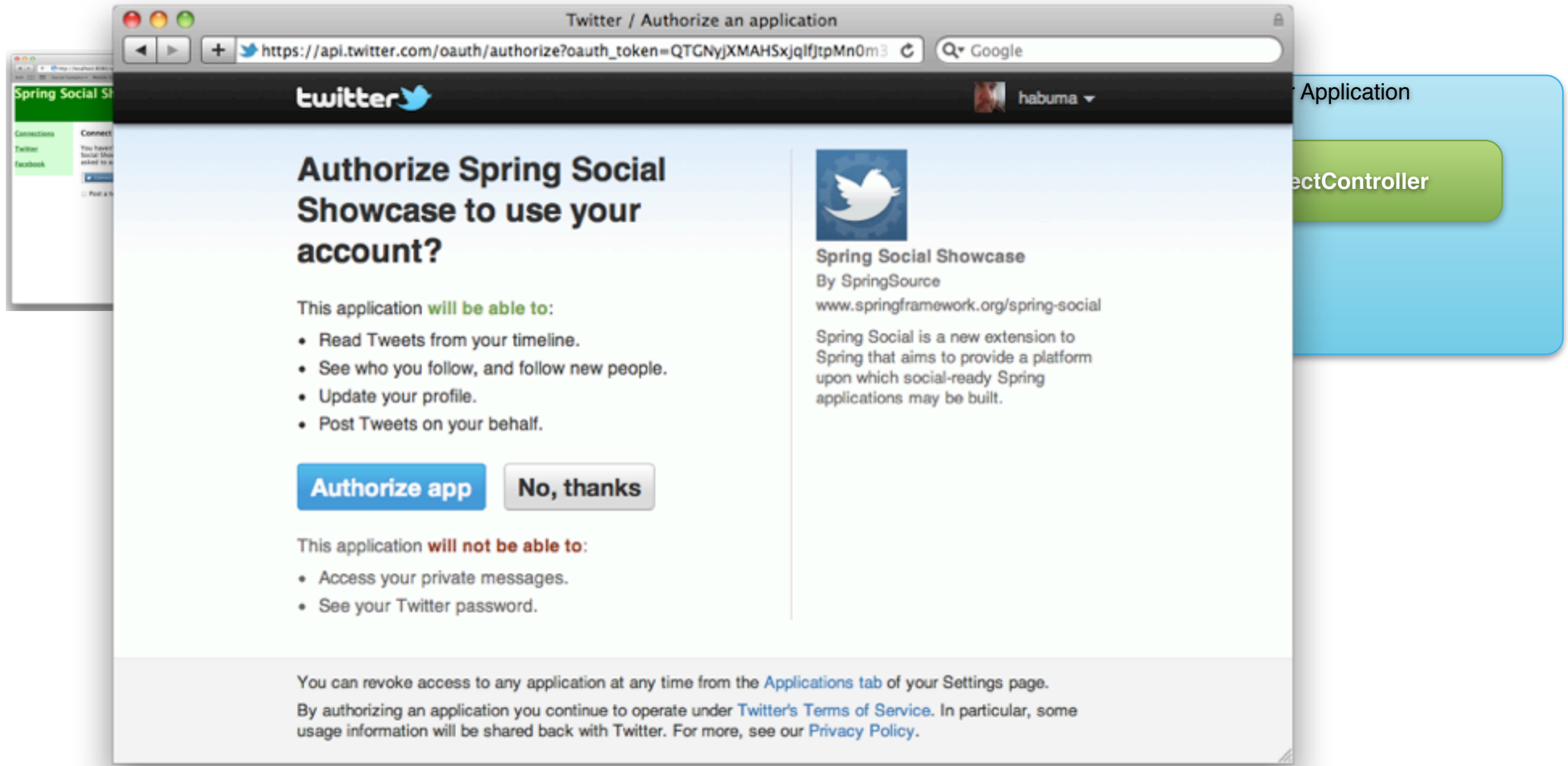




Service Provider
(Twitter, Facebook, etc)

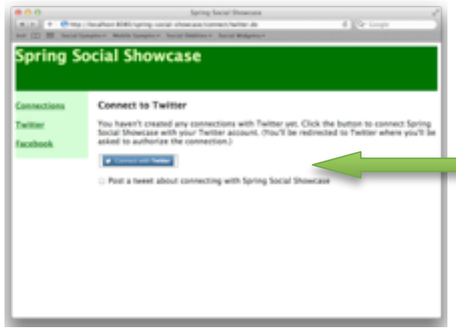
Redirect browser to provider's authorization page

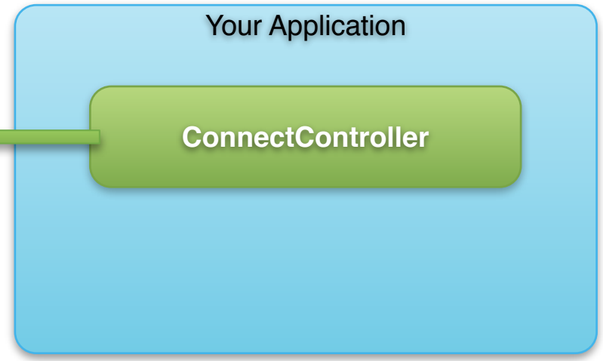
ConnectController Flow



Redirect browser to provider’s authorization page

ConnectController Flow

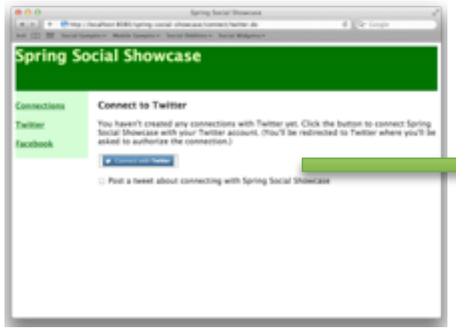




Service Provider
(Twitter, Facebook, etc)

Redirect browser to provider's authorization page

ConnectController Flow



GET /connect/{provider ID}?oauth_token={token}
GET /connect/{provider ID}?code={code}

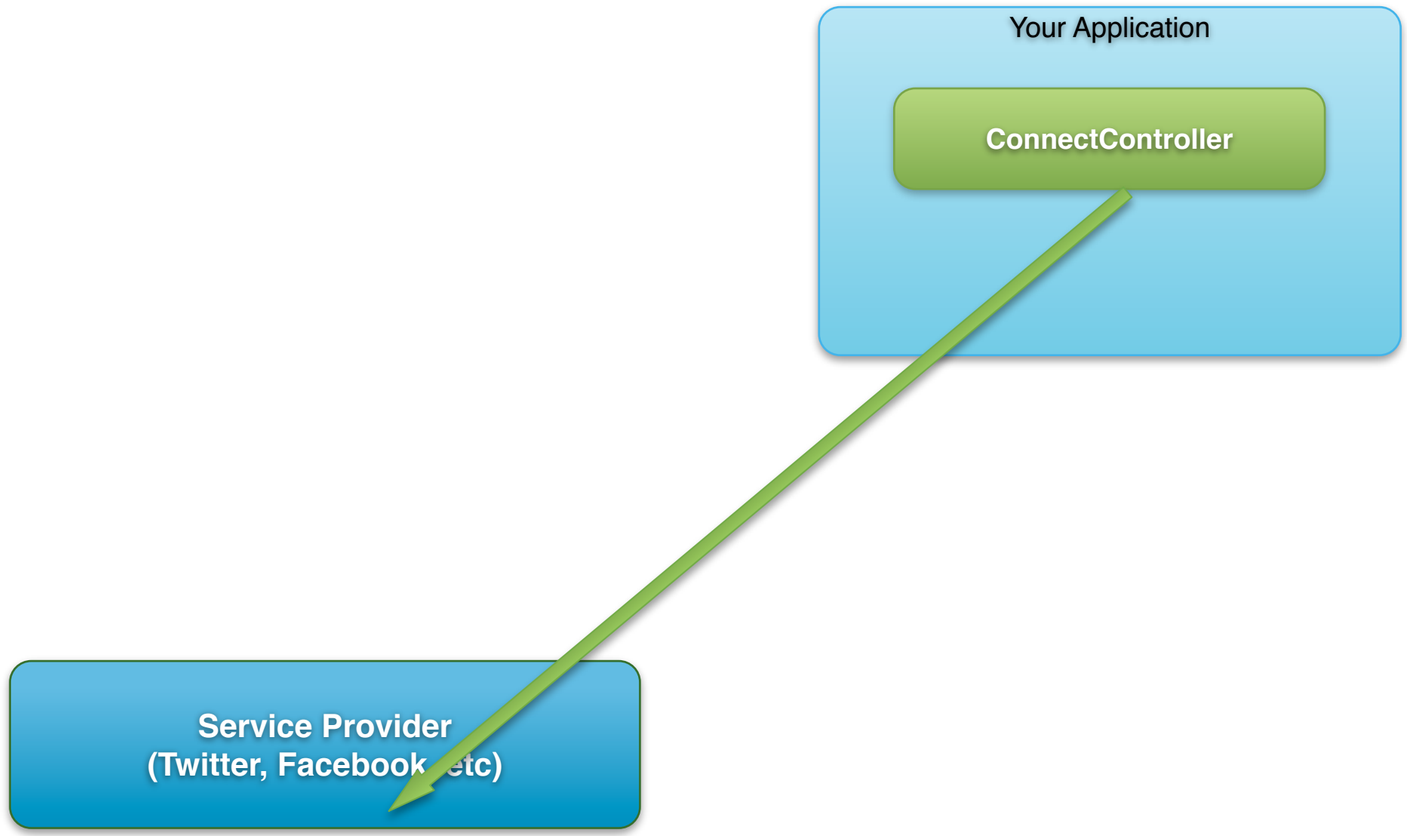
Your Application

ConnectController

Service Provider
(Twitter, Facebook, etc)

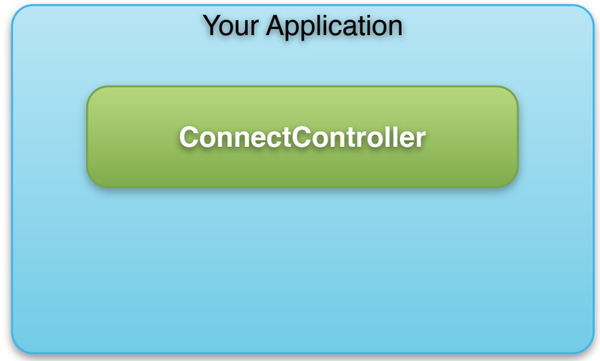
Provider redirects to callback URL

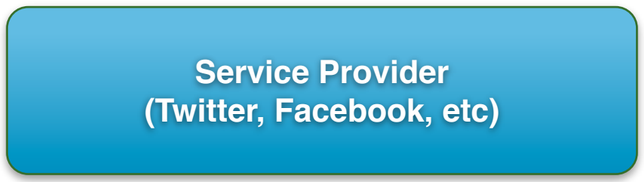
ConnectController Flow



Exchange request token and/or code for access token

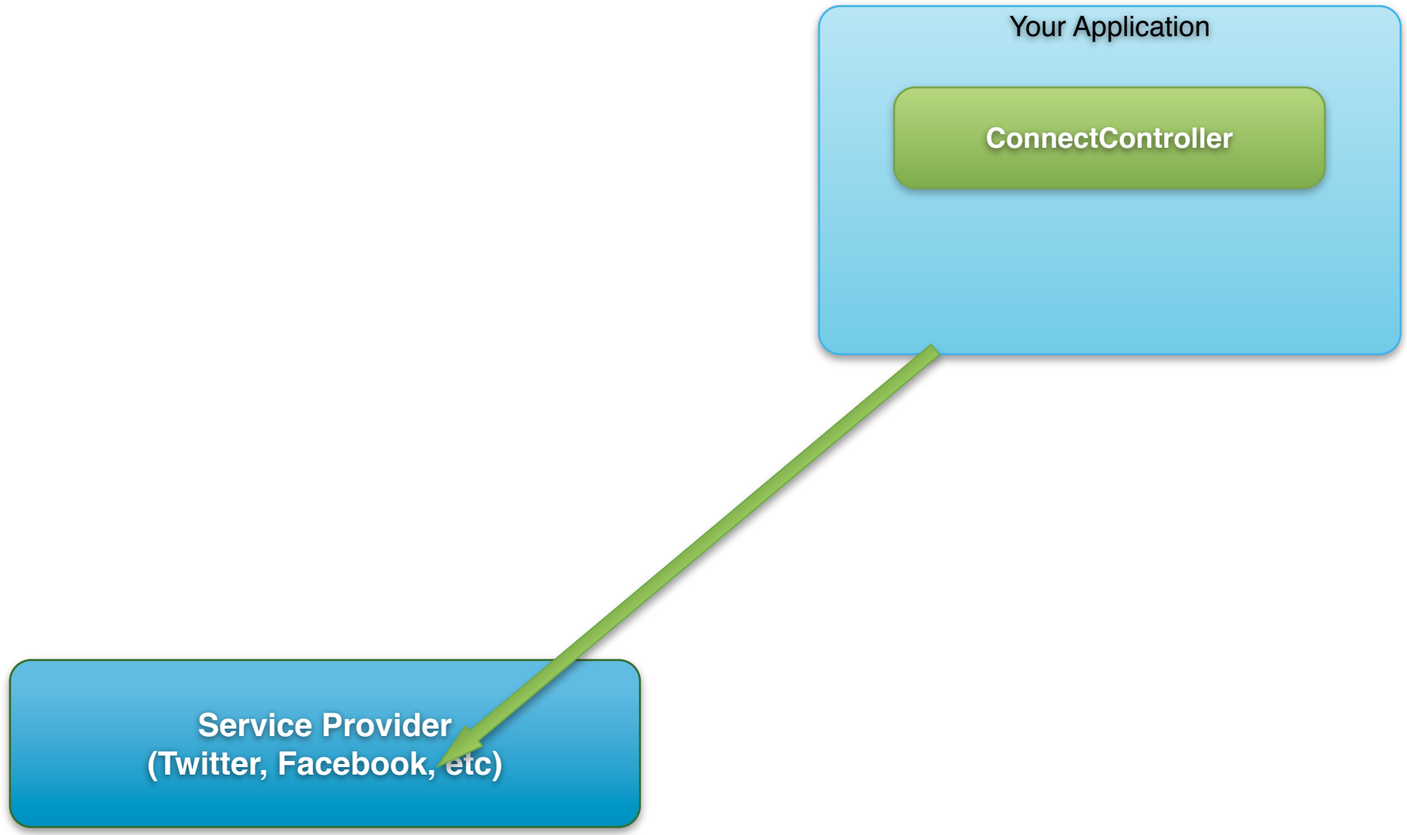
ConnectController Flow





ConnectController stores connection details in connection repository

ConnectController Flow



Application can make API calls via API binding

Connection Status Page View

```
<form action="<c:url value="/connect/twitter" />" method="POST">
  <div class="formInfo">
    <p>
    You haven't created any connections with Twitter yet.
    Click the button to connect with your Twitter account.
    </p>
  </div>
  <p>
    <button type="submit">
    <img src="<c:url value="/resources/social/twitter/connect-with-twitter.png" />"/>
    </button>
  </p>
</form>
```

Provider Sign In

- A convenience for users
- Enables authentication to an app using their connection as credentials
- Implemented with ProviderSignInController
- Works consistently with any provider

Configuration: ProviderSignInController

- Performs a similar flow as ConnectController
- Compares connections (by user ID)
- If there's a match, the user is signed into the application
- Otherwise, the user is sent to signup page
  - Connection is be established after signup

```
@Bean
public ProviderSignInController providerSignInController(
            RequestCache requestCache) {
  return new ProviderSignInController(connectionFactoryLocator(),
                        usersConnectionRepository(),
                        new SimpleSignInAdapter(requestCache));
}
```

Configuration: ProviderSignInController

- Performs a similar flow as ConnectController
- Compares connections (by user ID)
- If there's a match, the user is signed into the application
- Otherwise, the user is sent to signup page
  - Connection is be established after signup

```
@Bean
public ProviderSignInController providerSignInController(
            RequestCache requestCache) {
  return new ProviderSignInController(connectionFactoryLocator(),
                        usersConnectionRepository(),
                        new SimpleSignInAdapter(requestCache));
}
```

Configuration: ProviderSignInController

- Performs a similar flow as ConnectController
- Compares connections (by user ID)
- If there's a match, the user is signed into the application
- Otherwise, the user is sent to signup page
  - Connection is be established after signup

```
@Bean
public ProviderSignInController providerSignInController(
            RequestCache requestCache) {
  return new ProviderSignInController(connectionFactoryLocator(),
                    usersConnectionRepository(),
                    new SimpleSignInAdapter(requestCache));
}
```

Configuration: ProviderSignInController

- Performs a similar flow as ConnectController
- Compares connections (by user ID)
- If there's a match, the user is signed into the application
- Otherwise, the user is sent to signup page
  - Connection is be established after signup

```
@Bean
public ProviderSignInController providerSignInController(
            RequestCache requestCache) {
  return new ProviderSignInController(connectionFactoryLocator(),
                        usersConnectionRepository(),
                        new SimpleSignInAdapter(requestCache));
}
```

ProviderSignInController Endpoints

- **POST /signin/{provider}**
  - Initiates the authorization flow, redirecting to the provider
- **GET /signin/{provider}?oauth_token={token}**
  - Handles an OAuth 1 callback
- **GET /signin/{provider}?code={authorization code}**
  - Handles an OAuth 2 callback
- **GET /signin**
  - Handles a callback when no oauth token or code is sent
  - Likely indicates that the user declined authorization

Spring Security OAuth

Spring Security OAuth

- **Extension to Spring Security**
  - originally a community contribution (now officially part of the project)
- **Features...**
  - endpoint management for OAuth service types
  - (along with corresponding client management)
  - token management (persistence, authentication)
  - integrations for the web, as well as through standard Spring Security
- **springsource.org/spring-security/oauth**

Setup Spring Security with Spring MVC...

```
<http access-denied-page="/login.jsp"
    xmlns="http://www.springframework.org/schema/security">

        <form-login authentication-failure-url="/login.jsp"
            default-target-url="/index.jsp"
            login-page="/login.jsp"
            login-processing-url="/login.do" />

        <logout logout-success-url="/index.jsp" logout-url="/logout.do" />

        <anonymous />

        <intercept-url pattern="/sparklr/**" access="ROLE_USER" />

        <custom-filter ref="oauth2ClientFilter" after="EXCEPTION_TRANSLATION_FILTER" />

</http>
```

Then Tell Spring Security About Our OAuth Endpoints

```
<authentication-manager
    xmlns="http://www.springframework.org/schema/security">
        <authentication-provider>
                <user-service>
                <user name="marissa" password="wombat"
                        authorities="ROLE_USER" />
                <user name="sam" password="kangaroo"
                        authorities="ROLE_USER" />
                </user-service>
        </authentication-provider>
</authentication-manager>
```

```
<oauth:client id="oauth2ClientFilter"
    redirect-on-error="${redirectOnError:false}" />

<oauth:resource id="sparklr"
    type="authorization_code" client-id="tonr"
    client-secret="secret" access-token-uri="${accessTokenUri}"
    user-authorization-uri="${userAuthorizationUri}" scope="read" />
```

Then get an Instance of a RestTemplate for the client...

```
<bean class="org.springframework.security.oauth2.client.OAuth2RestTemplate">
  <constructor-arg ref = “sparklr”/>
</bean>
```

And The REST of it

- **Spring Data REST -** exposes (JPA, MongoDB, GemFire, Neo4J, Redis) repositories built on Spring Data repositories
- **Spring HATEOAS -** take your REST-fu to the next level with support for HTTP as the engine of application state

@starbuxman | josh.long@springsource.com

http://slideshare.net/joshlong

Questions?