Summer MVC 1.0 – Documentation

1. Overview
2. Get Started
3. Get Started with Javache Embedded.
4. Summer MVC Components
5. Routing
6. Dependency Container and Scanning
7. Template Engine (jTwig)
8. Validation, Binding
9. Security

Overview

Summer MVC, unlike Spring, is a created more like a library and not a platform. Therefore if you want to use Summer MVC along with **Javache** you will have to put the jar file in the “lib” folder. The compiling is kept simple (just run mvn package). Inside Summer MVC you will find other third party libraries like Gson for converting objects into JSON and JTwig – The chosen templating engine for Summer MVC.

After importing the framework, you will be granted with annotations such as @Controller and @Service which will be the focal point of this documentation.

# Get Started

If you have experience with Spring MVC, you will find it **REALLY** similar as the names of most of the annotations and interfaces are the exact same.

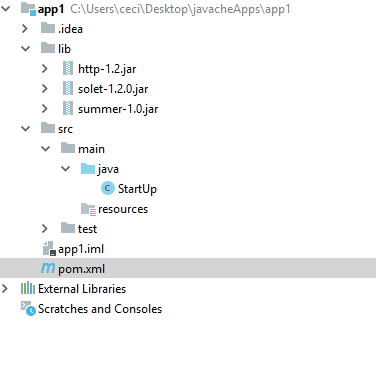
So to get started, you need to know how **Broccolina** works. Broccolina extracts every .jar file that was present in the webapps folder during the initial start of the server. Then scans the extracted folder for .class files and filters those, who are assignable from BaseHttpSolet. Then those files are being loaded and executed if a route for a particular HttpSolet is present.

So therefore if we want our app to be detected, we need to have at least one HttpSolet. Summer MVC has only one HttpSolet which listens on (“/\*”) which basically means everything.

But we cannot use that solet because we put the Summer MVC .jar file in the lib folder and not in the webapps folder. So to run out application we want to extend that Summer MVC solet and therefore inherit all the logic from the Summer MVC .jar file since the framework is made in such way what all the flow passes through the “Solet Dispatcher”.

Okay, lets run the app.

First create a new maven project, select java 11 and proceed.

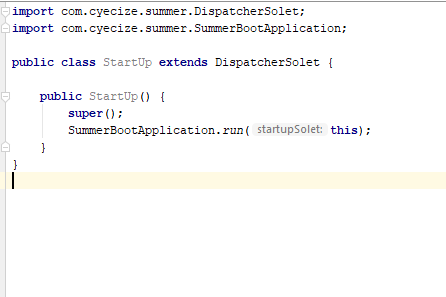


We want to create a folder named lib where we will include the 3 mandatory dependencies for out app to compile and run properly. Those are Http, SoletApi and Summer MVC.

Here is an example pom.xml for loading those dependencies.

<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>summer</**artifactId**>  
 <**scope**>system</**scope**>  
 <**version**>1.0</**version**>  
 <**systemPath**>${basedir}/lib/summer-1.0.jar</**systemPath**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>http</**artifactId**>  
 <**scope**>system</**scope**>  
 <**version**>1.2</**version**>  
 <**systemPath**>${basedir}/lib/http-1.2.jar</**systemPath**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>solet</**artifactId**>  
 <**scope**>system</**scope**>  
 <**version**>1.2.0</**version**>  
 <**systemPath**>${basedir}/lib/solet-1.2.0.jar</**systemPath**>  
</**dependency**>

Then You want to create a class and extend DispatcherSolet and inside the constructor call SummerBootApplication.run(this);



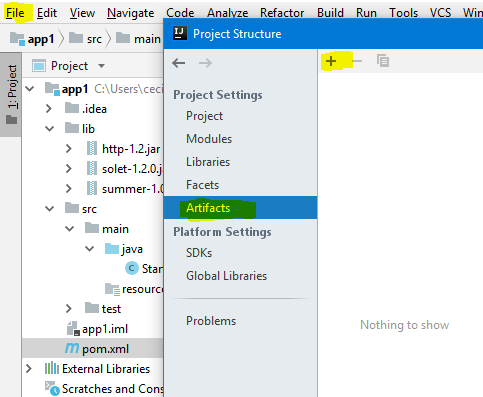
That’s it, now your application will be recognized by Broccolina. But to run the app we also need to package It in the proper way, which is:

classes folder what contains the compile output.

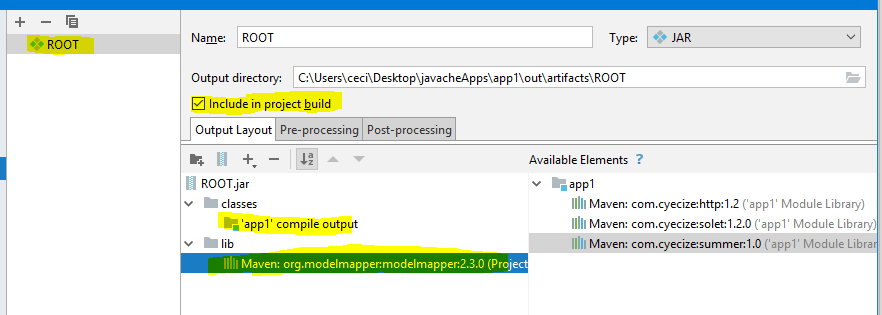
lib folder to place your third party libraries.

**NOTE** that you can put your libraries in the lib folder in Javache and it will still work!

So to create a proper jar , in IntelliJ go to file -> Project Structure -> Artifacts and click on the plus sign



Then give the jar file name “ROOT” since root is the default app in Javache and make sure you check include in project build.



You can see that I have added ModelMapper as an external library but left Http, SoletAPI and

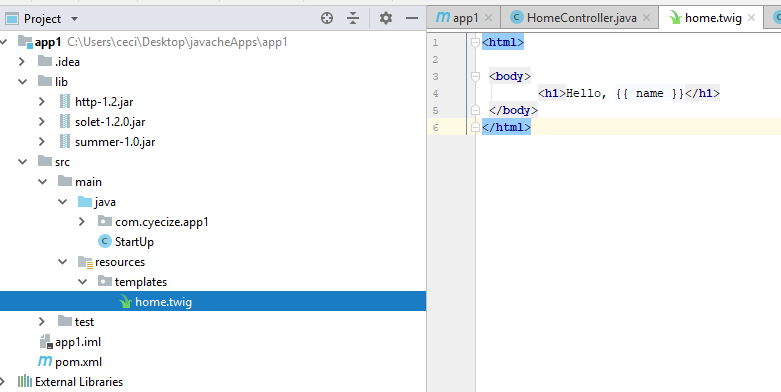
Summer MVC outside because they will be in the javache’s lib folder.

With that the setup is ready. Not lets create a simple page.

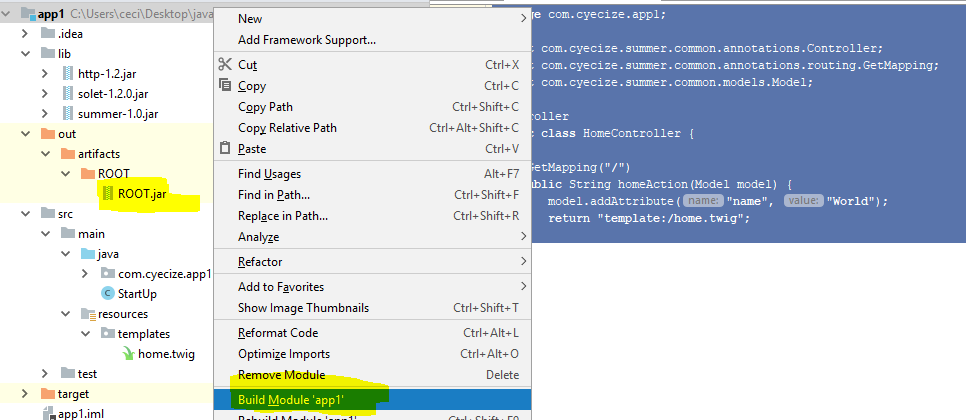
Create a controller HomeController, annotate it with @Controller and add a public String homeAction, annotated with @GetMapping.

**package** com.cyecize.app1;  
  
**import** com.cyecize.summer.common.annotations.Controller;  
**import** com.cyecize.summer.common.annotations.routing.GetMapping;  
**import** com.cyecize.summer.common.models.Model;  
  
@Controller  
**public class** HomeController {  
  
 @GetMapping(**"/"**)  
 **public** String homeAction(Model model) {  
 model.addAttribute(**"name"**, **"World"**);  
 **return "template:/home.twig"**;  
 }  
}

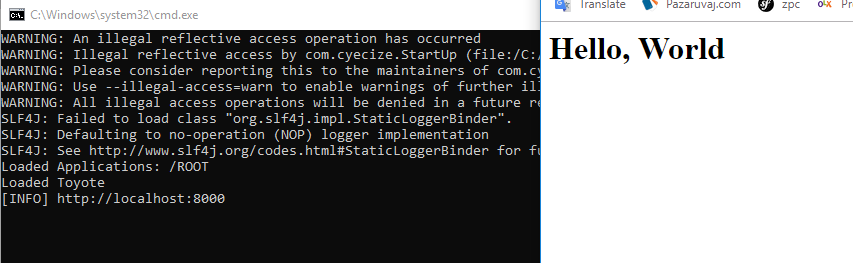
After that in your resources folder, add new folder and name it templates. Inside create a file named home.twig.



Build the app and get the ROOT.jar and paste it into Javache’s webapps.



And then run the app.



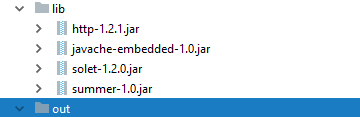
Voilà!

# Get Started With Javache Embedded

The above mentioned method is good, but it is more to the production side of things.

While developing your application you are better off using the embedded version of Javache. The embedded version of javache is a bundle of the whole web server with exception of HttpSolet and Http apis.

So in order to include Javache embedded to your app, include the following dependencies.



The pom dependencies should look like this.

<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>javache-embedded</**artifactId**>  
 <**version**>1.0</**version**>  
 <**scope**>system</**scope**>  
 <**systemPath**>${basedir}/lib/javache-embedded-1.0.jar</**systemPath**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>http</**artifactId**>  
 <**version**>1.2.1</**version**>  
 <**scope**>system</**scope**>  
 <**systemPath**>${basedir}/lib/http-1.2.1.jar</**systemPath**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>solet</**artifactId**>  
 <**version**>1.2.0</**version**>  
 <**scope**>system</**scope**>  
 <**systemPath**>${basedir}/lib/solet-1.2.0.jar</**systemPath**>  
</**dependency**>  
  
<**dependency**>  
 <**groupId**>com.cyecize</**groupId**>  
 <**artifactId**>summer</**artifactId**>  
 <**version**>1.0</**version**>  
 <**scope**>system</**scope**>  
 <**systemPath**>${basedir}/lib/summer-1.0.jar</**systemPath**>  
</**dependency**>

To run the application, you need to again extend DispatcherSolet and call SummerBootApplication.run, but now we add a main method where we call JavacheEmbedded.startServer….

**public class** StartUp **extends** DispatcherSolet {  
 **public** StartUp() {  
 SummerBootApplication.*run*(**this**);  
 }  
  
 **public static void** main(String[] args) {  
 JavacheEmbedded.*startServer*(8000, StartUp.**class**);  
 }  
}

This will grant you with a web server, embedded in your application for debugging options and faster build time and overall build process.

Once you are ready with the development, you can package the app the same way as the above mentioned example, excluding Javache embedded and your project will be ready to go.

# Summer MVC Components

The main components are:

Bean, Service, Component, Controller.

Where beans are with the highest priority while scanning and components and controllers are with the least. A service can inject a bean and/or another service.Components and Controllers can inject both services and beans but not controllers and components.

Bean – Beans, similar to Spring, are used to load a service that is not a part of the app. To load beans, create a class and annotate it with

com.cyecize.summer.common.annotations.BeanConfig

Make sure that the class has empty constructor.

Make sure @Bean methods are not void and it do NOT take any parameters.

@BeanConfig  
**public class** BeanCfg {  
  
 **public** BeanCfg() {}  
  
 @Bean  
 **public** ModelMapper getModelMapper() {  
 **return new** ModelMapper();  
 }  
}

Servcie – Services are important for a well-structured code and they are fully supported in Summer MVC. A service is a class that might or might not implement an interface but it does have a @Service annotation.

@Service  
**public class** ExampleServiceImpl {  
  
 **private final** ModelMapper **modelMapper**;  
  
 **public** ExampleServiceImpl(ModelMapper modelMapper) {  
 **this**.**modelMapper** = modelMapper;  
 System.***out***.println(**"i was created!"**);  
 }  
}

In this example, ModelMapper will be injected since we have created a bean for it. In case of no suitable service or bean, Summer will throw an exception. Be careful not to fall into an unresolvable satiation by having one service depend on another service which depends on the first service.

Component – Components are like extensions to Summer. Those can be Interceptors, Custom validators and others, but we will talk about those later. The important thing is that components also get Dependency Injection.

Controller –The class where we write our action methods. Also gets Dependency Injection.

**@Service, @Component, @Controller** annotations have lifespan property, which can be set to the default Singleton or Request which means that the particular class will be re-instantiated on every request.

Those components also have an option for a method with **@PostConstruct** annotation, which will be executed after the object has been instantiated.

Custom Interceptor:   
To create a custom interceptor, you need to create a class, implement InterceptorAdaper and annotate the class with @Component.

**import** com.cyecize.solet.HttpSoletRequest;  
**import** com.cyecize.solet.HttpSoletResponse;  
**import** com.cyecize.summer.common.annotations.Component;  
**import** com.cyecize.summer.common.extensions.InterceptorAdapter;  
**import** com.cyecize.summer.common.models.Model;  
  
@Component  
**public class** CustomInterceptor **implements** InterceptorAdapter {  
  
 @Override  
 **public boolean** preHandle(HttpSoletRequest request, HttpSoletResponse response, Object handler) {  
  
 *//decide if you want to let the request continue...* **if** (request.getRequestURL().equalsIgnoreCase(**"/forbidden"**)) {  
 **return false**;  
 }  
 **return true**;  
 }  
  
 @Override  
 **public void** postHandle(HttpSoletRequest request, HttpSoletResponse response, Object handler, Model model) {  
 *//do things such as logging or redirecting here.* }  
}

# Routing

## @GetMapping, @PostMapping, @RequestMapping

are the annotations that are used for initializing a route.

Both annotations have **produces** method which specifies the Content-Type of the response.

### @RequestMapping

annotation can be used on a controller which will result in a base route.

For example if we have “/users” as a base route, all action methods inside that controller will have “/users” prepended.

## Path variables

are also supported. “/user/{id}/details” as an example has one path variable – id. We can access that variable by specifying a parameter annotated with **@PathVariable(“id”) int id.**   
Summer MVC will parse the string into a number for us and if it fails it will return Integer.minValue.

If you annotate the parameter with **@ConvertedBy** annotation and specify a data adapter, you can convert the parameter to any object that you want (see Custom Data Adapters for more information).

@GetMapping(value = **"/user/{username}/details"**, produces = **"application/json"**)  
**public** String userDetails(@PathVariable(**"username"**) String name) {  
 **return "{'name':'"** + name + **"'}"**;  
}

## Request Parameters

can be accessed directly by annotating a parameter with **@RequestParam(“paramName”)** long param….

@RequestParam first looks for query parameters and then it looks for body parameters if available.  
All primitive data types are supported. Custom objects are also supported If you annotate the parameter with @ConvertedBy (see Custom Data Adapters for more information).

@GetMapping(**"/example/test"**)  
**public** JsonResponse exampleAction(@RequestParam(**"size"**) **long** size, @RequestParam(**"name"**) String name) {  
 JsonResponse response = **new** JsonResponse();  
 response.addAttribute(**"size"**, size);  
 response.addAttribute(**"name"**, name);  
 **return** response;  
}

## Return Types

In Summer MVC you can return: String, Model, ModelAndView, JsonResponse, other object.

**If you return String** you can specify a view by returning “template:/home/index.twig”.  
To redirect – “redirect:/location” – location is relative to the current app name.  
So if your app name is app3 the home route will be on localhost:8000/app3 and if you return “redirect:/”, you will be redirected to localhost:8000/app3/.

If you don’t specify view or redirect the output will be written on the page.

**If you return Model** you can specify the view by calling model.addAttribute(“view”, “view.twig”) or redirect by model.addAttribute(“view”, “redirect:/contacts”)

@GetMapping(**"/"**)  
**public** Model homeAction(Model model) {  
 model.addAttribute(**"name"**, **"World"**);  
 model.addAttribute(**"view"**, **"redirect:/user/aehaeh/details"**);  
 **return** model;  
}

Make sure you accept the model as a param because this will sync the parameters with other return types.

**If you return ModelAndView** for example, you can still use the model to add parameters for the view, but return the modelAndView in the end.

@GetMapping(**"/"**)  
**public** ModelAndView homeAction(Model model) {  
 model.addAttribute(**"name"**, **"World"**);  
 **return new** ModelAndView(**"/home.twig"**);  
}

This method is useful when we work with validations, but more on that later. You can still call modelAndView.addObject(“paramName”, value);

**If you return JsonResponse** the object will be parsed into json by Gson and the content type will be automatically set to application/json.

**Finally if you return another object**, Gson will parse it but it is up to you to specify the Content-Type.

There are many parameters that you can accept on your action route but those will be mentioned in the Scanning and Dependency Container section.

## Exception handling

If no mathching route is found, an exception will be thrown **com.cyecize.summer.areas.routing.exceptions.HttpNotFoundException which you can catch by an @ExceptionListener**

@ExceptionListener(HttpNotFoundException.**class**)  
**public** String notFoundErr(HttpNotFoundException ex, Model model) {  
 model.addAttribute(**"ex"**, ex);  
 **return "template:/404.twig"**;  
}

<**html**>  
<**body**>  
<**h1**>Ex is {{ ex.message }}</**h1**>  
</**body**>  
</**html**>

Exceptions listener is just like a route but it listens for a certain exception.  
You can make a global exception listener by listening for Throwable but it is not recommended.  
Exceptions are sorted by hierarchy so you can place them wherever you want that they will still not override.

# Dependency Container & Scanning

The way that scanning works in Summer MVC goes as follows:

After we call SummerBootApplication.run(this) a scanning service is created with a scanning path equal to the path of the parameter “this”. Then the scanner looks for .class files and adds them to a collection of classes. Then other services such as BeanLoadingService and ServiceLoadingService take those classes, filter those who have their needed annotations and then create instance of the remaining classes.  
Finally all loaded services and beans are loaded in the dependency container.  
But there are some more classes that are present In the dependency container, but are instantiated from the platform on every request instead of the user creating them. Those are:  
**HttpSoletRequest  
HttpSoletResponse  
SoletConfig  
HttpSession  
RenderingService (Twig)  
Model  
ModelAndView  
RedirectAttributes –** Redirected attributes will be populated in the **Model** on the next request. **Principal  
BindingReslt  
RedirectedBindingResult  
DependencyContainer** itself is also stored inside the dependency container.

If the method is an exception listener, the **exception** will also be present in the container.

The above stated platform objects can be accessed as a parameter in your action methods.

# Template Engine

Twig is the templating engine for Summer MVC as it is really easy to use and it packs more built in features like inheritance where Thymeleaf for example lacks.

The starting path for templates is resources/templates.  
You can render a template by:

1. *Returning a string “template:/nameOfTemplate.twig”, “template:nameOfTemplate.twig”*
2. *Returning a model with a param “view” and value to that param – “nameOfTemplate.twig”*
3. *Returning a ModelAndView with a the method setView(“nameOfTemplate.twig”) being called.*

JTwig builtIn functions

**{{ path(“/path”) }}** at first might seem to do nothing but it actually does quite a lot – adds an app prefix so if your app .jar file is different that ROOT.jar it will append its name before the route.  
so **app3.jar will be localhost:8000/app3/path instead of localhost:8000/path** which is not a valid URL in this case.

**{{ hasRole(“ROLE\_NAME”) }}** checks if the user has that particular role.

**{% if hasRole(‘ROLE\_ADMIN’) %} <p>Hello, admin </p> {% endif %}**

**{{ formErrors }} or {{ formErros(‘fieldName’) }}** returns a list of **FieldError**. An example use is:

{% for err in formErros(‘username’) %} {{ err.message }} {% endfor %}

**{{ user }}** is a global variable which keeps the logged in user if one exists.

**{{ url(“/route”) }}** does the same as {{ path }} but it appends the whole URI in front of the route.  
for example {{ url(“/login”) }} will result in //localhost:8000/login. This is very useful if you are sending emails and you want the users to refer to your website.

## Twig Globals

Summer MVC allows you to create a service that will be accessible in the templating engine at runtime by annotating a service with @TemplateService

@Service  
@TemplateService(serviceNameInTemplate = **"util"**)  
**public class** TemplateHelper {  
   
 **public** String dateFormat(LocalDateTime localDateTime) {  
 *//format your date and return the result* }  
}

After you create the service, it will be accessible in twig like so:

{{ util.dateFormat(localDateTimeObject) }}

**{{ user }} and {{ request }}** are globals where user is the Principal and request is the HttpSoletRequest.

# Validation, Binding

Summer MVC has support for all primitive data types, Lists and files.

**package** com.example.appl.bindingModels;  
  
**import** com.cyecize.summer.areas.routing.interfaces.MultipartFile;  
  
**import** java.util.List;  
  
**public class** TestBindingModel {  
   
 **private** String **name**;  
   
 **private int age**;  
   
 **private char sex**;  
   
 **private** MultipartFile **profileImage**;  
   
 **private** List<Integer> **listOfSubjectIds**;  
   
 **public** TestBindingModel() {  
   
 }  
}

@PostMapping(**"/register"**)  
**public** ModelAndView registerAction(TestBindingModel bindingModel) {  
 *//****todo use this binding model***}

The fields of the class **must match** the input fields in the HTML form.

If we add a string for an age, we will get the min val for int. That applices to all primitive types.  
Lists can also have any primitive generic type including String. Summer MVC will parse the html values if possible.

That is really nice but there is no validation on this binding model.

## Validations

Luckily for us, Summer MVC provides validation functionality. You can create your custom validation or use on of the provided:

@NotNull – works on all objects, checks if the object is null  
@NotEmpty – works on strings and lists, checks if the size is > 0.  
@FieldMatch – perfect for checking if passwords match, checks if 2 fields are equal.  
@Min/MaxLength – works on Strings, collections, maps , Multipart files…. Performs a check for their length/size.  
@Min/Max works on Numbers, same as @Max/MinLength.  
@RegEx - works on strings by testing if the provided pattern matched the field val.  
@MediaType – works on Multipart Files. Checks if the file’s media type match any of the given.

@NotNull(message = **"Username cannot be empty."**)  
@MaxLength(length = 15, message = **"Username too long."**)  
**private** String **username**;  
  
@NotEmpty(message = **"Email is empty."**)  
**private** String **email**;  
  
@MinLength(length = 6, message = **"Password too short."**)  
**private** String **password**;  
  
@FieldMatch(fieldToMatch = **"password"**, message = **"Passwords did not match."**)  
**private** String **passwordConf**;  
  
**public** RegisterBindingModel() {  
  
}

To activate validation for a given binding model, annotate the parameter with @Valid

@PostMapping(**"/register"**)  
**public** ModelAndView registerPost(@Valid RegisterBindingModel bindingModel, BindingResult bindingResult, RedirectAttributes redirectAttributes) {  
   
 redirectAttributes.addAttribute(**"model"**, bindingModel);  
 **if** (bindingResult.hasErrors()) {  
 **return this**.redirect(**"/register"**);  
 }  
  
 **try** {  
 **this**.**userService**.createUser(bindingModel);  
 } **catch** (RuntimeException ex) {  
 **return this**.redirect(**"/register"**);  
 }  
 **return this**.redirect(**"/login"**);  
}

Here we use RedirectAttributes to send the registerBindingModel to the other route to avoid losing the data that the user sent us.

Here is the get method as well.

@GetMapping(**"/register"**)  
**public** ModelAndView registerGet(Model model) {  
 **if** (!model.hasAttribute(**"model"**)) {  
 model.addAttribute(**"model"**, **new** RegisterBindingModel());  
 }  
 **return** view(**"/security/register.twig"**);  
}

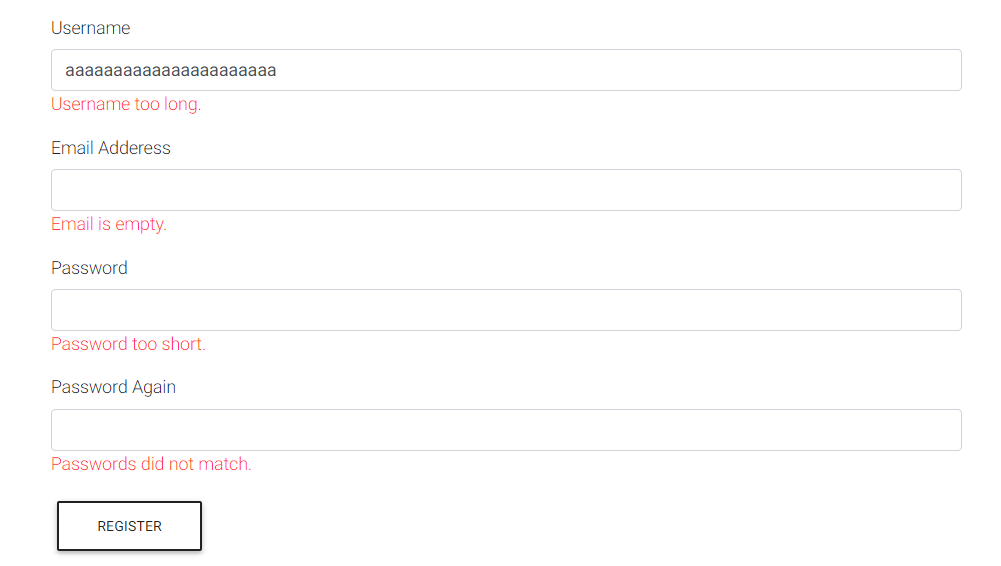
And here is the register form.

<**form method="post" action="**{{ path(**"/register"**) }}**"**>  
 <**div class="form-group"**>  
 <**label for="username"**>Username</**label**>  
 <**input type="text" name="username" id="username" value="**{{ model.username }}**" required class="form-control"**>  
 {% **embed '../field-error-handler.twig'** with {**'fieldName'**:**'username'**} %} {% **endembed** %}  
 </**div**>  
 <**div class="form-group"**>  
 <**label for="email"**>Email Adderess</**label**>  
 <**input type="email" name="email" id="email" value="**{{ model.email }}**" required class="form-control"**>  
 {% **embed '../field-error-handler.twig'** with {**'fieldName'**:**'email'**} %} {% **endembed** %}  
 </**div**>  
 <**div class="form-group"**>  
 <**label for="password"**>Password</**label**>  
 <**input type="password" name="password" id="password" required class="form-control"**>  
 {% **embed '../field-error-handler.twig'** with {**'fieldName'**:**'password'**} %} {% **endembed** %}  
 </**div**>  
 <**div class="form-group"**>  
 <**label for="pass-again"**>Password Again</**label**>  
 <**input type="password" name="passwordConf" id="pass-again" required class="form-control"**>  
 {% **embed '../field-error-handler.twig'** with {**'fieldName'**:**'passwordConf'**} %} {% **endembed** %}  
 </**div**>  
 <**div class="form-group"**>  
 <**button type="submit" class="btn btn-outline-dark"**>Register</**button**>  
 </**div**>  
</**form**>

That {% embed %} attribute lets you reuse code, here is what I am embedding.

{% **for** fieldErr **in** formErrors(fieldName) %}  
 <**p class="text-danger"**>{{ fieldErr.message }}</**p**>  
{% **endfor** %}

Lets see how our validation works.



How to create a custom validation… Lets say we want to validate whether that username is unique...

Create an annotation @Unique username that has a method message() which is mandatory.  
Annotate that annotation with @Constraint(validatedBy=TheNameOfYourValidator.class)

**import** com.cyecize.summer.areas.validation.annotations.Constraint;  
  
**import** java.lang.annotation.ElementType;  
**import** java.lang.annotation.Retention;  
**import** java.lang.annotation.RetentionPolicy;  
**import** java.lang.annotation.Target;  
  
@Retention(RetentionPolicy.***RUNTIME***)  
@Target(ElementType.***FIELD***)  
@Constraint(validatedBy = UniqueUserConstraint.**class**)  
**public** @**interface** UniqueUsername {  
 String message() **default "Username is not unique"**;  
}

The validator must implement ConstraintValidator<Annotation, FieldType> and be annotated with @Component

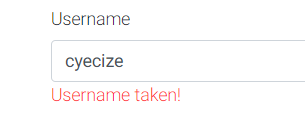
**import** com.cyecize.summer.areas.validation.interfaces.ConstraintValidator;  
**import** com.cyecize.summer.common.annotations.Component;  
**import** com.example.appl.services.UserService;  
  
@Component  
**public class** UniqueUserConstraint **implements** ConstraintValidator<UniqueUsername, String> {  
  
 **private final** UserService **userService**;  
  
 **public** UniqueUserConstraint(UserService userService) {  
 **this**.**userService** = userService;  
 }  
  
 @Override  
 **public boolean** isValid(String s, Object o) {  
 **if** (s == **null**) {  
 **return true**;  
 }  
 **return this**.**userService**.findOneByUsername(s) == **null**;  
 }  
}

This is an example where I call the db and check if that user exists.   
Dependency injection is available here!

Now the binding model.

@NotNull(message = **"Username cannot be empty."**)  
@MaxLength(length = 15, message = **"Username too long."**)  
@UniqueUsername(message = **"Username taken!"**)  
**private** String **username**;

And the result is…



You can also add custom errors new calling **BindingResult.addNewError(FieldError);**

## Custom Data Adapters

Lets say you want to create a product which belongs to a category. Normally you pass the name or the ID of that category and then locate the category with the given data.

Summer MVC lets you map the category entity directly in the binding model and by doing so achieving better validation.

You need to create a @Component annotated class that implements the DataAdapter interface.

@Component  
**public class** IdToCategoryAdapter **implements** DataAdapter<Category> {  
  
 **private final** CategoryService **categoryService**;  
  
 **public** IdToCategoryAdapter(CategoryService categoryService) {  
 **this**.**categoryService** = categoryService;  
 }  
  
 @Override  
 **public** Category resolve(String paramName, HttpSoletRequest httpSoletRequest) {  
  
 **try** {  
 **long** id = Long.*parseLong*(httpSoletRequest.getBodyParameters().get(paramName));  
 **return this**.**categoryService**.findOneById(id);  
 } **catch** (Exception ignored) {  
 }  
  
 **return null**;  
 }  
}

Then in your bindingModel you can say:

@NotNull(message = ***FIELD\_CANNOT\_BE\_EMPTY***)  
@ConvertedBy(IdToCategoryAdapter.**class**)  
**private** Category **category**;

**Using this method you can also resolve even collections and maps…**

@Component  
**public class** ChannelCategoryListDataAdapter **implements** DataAdapter<List<ChannelCategory>> {  
  
 **private final** CategoryService **categoryService**;  
  
 **public** ChannelCategoryListDataAdapter(CategoryService categoryService) {  
 **this**.**categoryService** = categoryService;  
 }  
  
 @Override  
 **public** List<ChannelCategory> resolveField(Field field, HttpSoletRequest httpSoletRequest) {  
 List<ChannelCategory> categories = **new** ArrayList<>();  
 Map<String, List<String>> bodyParametersAsList = httpSoletRequest.getBodyParametersAsList();  
  
 **if** (!bodyParametersAsList.containsKey(field.getName())) {  
 **return** categories;  
 }  
  
 **for** (String catName : bodyParametersAsList.get(field.getName())) {  
 ChannelCategory channelCategory = **this**.**categoryService**.findOneByName(catName);  
 **if** (channelCategory != **null** && !categories.contains(channelCategory)) {  
 categories.add(channelCategory);  
 }  
 }  
  
 **return** categories;  
 }  
}

### @ConvertedBy annotation

If you have multiple data adapters for the same object, you can use this annotation to specify exactly which data adapter you want to handle your request.

@NotNull  
@ConvertedBy(LongToUserConverter.**class**)  
**private** User **user**;

This helpful if you have data adapter that uses the user id and the other uses the username.

# Security

Summer MVC also packs a Security Bundle.

If you want your app to use Summer MVC security, you need to provide a few parameters.

One of them is SecurityConfig class that contains information about security routes.  
You have to create a bean for it.

@Bean  
**public** SecurityConfig getSecurityConfig() {  
 **return new** SecurityConfig()  
 .setLoginURL(**"/login"**)  
 .setLogoutURL(**"/logout"**)  
 .setUnauthorizedURL(**"/unauthorized"**)  
 .setLogoutRedirectURL(**"/"**);  
}

The second thing is the UserDetails and GrantedAuthority interfaces which represent your user and his roles.

Role entity:

@Entity  
@Table(name = **"roles"**)  
**public class** Role **implements** GrantedAuthority {  
  
 @Id  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 @Column(name = **"id"**, updatable = **false**, unique = **true**, nullable = **false**)  
 **private** Long **id**;  
  
 @Column(name = **"role\_type"**, unique = **true**, nullable = **false**)  
 **private** String **roleType**;  
  
 @ManyToMany(targetEntity = User.**class**, mappedBy = **"roles"**, fetch = FetchType.***EAGER***)  
 **private** List<User> **users**;  
  
 **public** Role() {  
 **this**.**users** = **new** ArrayList<>();  
 }  
  
 **public** Long getId() {  
 **return id**;  
 }  
  
 **public void** setId(Long id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getRoleType() {  
 **return roleType**;  
 }  
  
 **public void** setRoleType(String roleType) {  
 **this**.**roleType** = roleType;  
 }  
  
 **public** List<User> getUsers() {  
 **return users**;  
 }  
  
 **public void** setUsers(List<User> users) {  
 **this**.**users** = users;  
 }  
  
 @Override  
 @Transient  
 **public** String getAuthority() {  
 **return this**.**roleType**;  
 }  
}

User Entity:

@Entity  
@Table(name = **"users"**)  
**public class** User **implements** UserDetails {  
  
 @Id  
 @Column(name = **"id"**, nullable = **false**, unique = **true**, updatable = **false**)  
 @GeneratedValue(strategy = GenerationType.***IDENTITY***)  
 **private** Long **id**;  
  
 @Column(name = **"username"**, nullable = **false**, unique = **true**)  
 **private** String **username**;  
  
 @Column(name = **"email"**, nullable = **false**, unique = **true**)  
 **private** String **email**;  
  
 @Column(name = **"password"**, nullable = **false**)  
 **private** String **password**;  
  
 @ManyToMany(targetEntity = Role.**class**, fetch = FetchType.***EAGER***)  
 @JoinTable(name = **"users\_roles"**,  
 joinColumns = @JoinColumn(name = **"user\_id"**, referencedColumnName = **"id"**),  
 inverseJoinColumns = @JoinColumn(name = **"role\_id"**, referencedColumnName = **"id"**))  
 **private** List<GrantedAuthority> **roles**;  
  
 **public** User() {  
 **this**.**roles** = **new** ArrayList<>();  
 }  
  
 **public** Long getId() {  
 **return id**;  
 }  
  
 **public void** setId(Long id) {  
 **this**.**id** = id;  
 }  
  
 **public** String getUsername() {  
 **return username**;  
 }  
  
 **public void** setUsername(String username) {  
 **this**.**username** = username;  
 }  
  
 **public** String getEmail() {  
 **return email**;  
 }  
  
 **public void** setEmail(String email) {  
 **this**.**email** = email;  
 }  
  
 **public** String getPassword() {  
 **return password**;  
 }  
  
 @Override  
 @Transient  
 **public** Collection<GrantedAuthority> getAuthorities() {  
 **return this**.**roles**;  
 }  
  
 **public void** setPassword(String password) {  
 **this**.**password** = password;  
 }  
  
 **public** List<GrantedAuthority> getRoles() {  
 **return roles**;  
 }  
  
 **public void** setRoles(List<GrantedAuthority> roles) {  
 **this**.**roles** = roles;  
 }  
  
 @Transactional  
 **public void** addRole(Role role) {  
 **if** (**this**.**roles**.contains(role)) {  
 **return**;  
 }  
 role.getUsers().add(**this**);  
 **this**.**roles**.add(role);  
 }  
}

After you have those implemented we can proceed with the provided functionality.  
**@PreAuthorize** annotation can be used on methods and on controllers where methods have priority.  
@PreAuthorize(AuthorizationType.***ANONYMOUS***)  
@PreAuthorize(role = **"ROLE\_ADMIN"**)

The first annotation states that the controller/action is ONLY anonymous and the second states that the user must be authenticated with role ‘ROLE\_ADMIN’.

If the user is logged in but lacks the role, an exception is thown, which you can catch with @**ExceptionListener**.

If the user it anonymous, he is first sent to the login page with a query param ?callback=/theReferingRoute. That way you can redirect the user back when he is logged in.

Here is an example of a login action

@GetMapping(**"/login"**)  
**public** ModelAndView loginAction(Model model, HttpSoletRequest request) {  
 String callback = request.getQueryParameters().get(**"callback"**);  
 **if** (callback == **null**) callback = **"/"**;  
 **return this**.view(**"security/login.twig"**, **"callback"**, callback);  
}  
  
@PostMapping(**"/login"**)  
**public** ModelAndView loginPost(Principal principal, LoginBindingModel bindingModel, RedirectAttributes redirectAttributes) {  
 User user = **this**.**userService**.findOneByUsernameOrEmail(bindingModel.getUsername());  
 redirectAttributes.addAttribute(**"model"**, bindingModel);  
 **if** (user == **null**) {  
 redirectAttributes.addAttribute(**"error"**, **"Invalid username or email"**);  
 **return this**.redirect(**"/login"**);  
 }  
  
 **if** (!BCrypt.*checkpw*(bindingModel.getPassword(), user.getPassword())) {  
 redirectAttributes.addAttribute(**"error"**, **"Wrong password mate"**);  
 **return this**.redirect(**"/login"**);  
 }  
  
 principal.setUser(user);  
 System.***out***.println(**"callback: "** + bindingModel.getCallback());  
 **return this**.redirect(bindingModel.getCallback());  
}

I am keeping the callback in the bindingModel so if the user does not get his password on the first try the callback is still there.

We login by accepting Princial as a param and calling principal.setUser(UserDetailsImplementation);  
logout is handled automatically!

After we login, twig will have access to the user variable.

**Secured Areas**Some folders like META-INF should not be accessed from users so Summer MVC provides Secured Areas.

@Bean  
**public** SecurityConfig getSecurityConfig() {  
 **return new** SecurityConfig()  
 .setLoginURL(**"/login"**)  
 .setLogoutURL(**"/logout"**)  
 .setLogoutRedirectURL(**"/"**)  
 .setUnauthorizedURL(**"/unauthorized"**)  
 .addSecuredArea(**new** SecuredArea(**"/static"**, **"ROLE\_ADMIN"**));  
}

By default “/META-INF” is added.

You need to call addSecuredArea and pass a new SecuredArea object.

The route you pass will be converted into regex pattern in the format:  
**^ + userRegex + .\***

So the route you provide is not constant but it is valid for sub directories/routes as well.

The role is a string which represents the getAuthority of the GrantedAuthority interface.  
**NOTE** that you can put non-existent role as well. That way nobody will have access to that directory.