Summer MVC 1.0 – Documentation

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

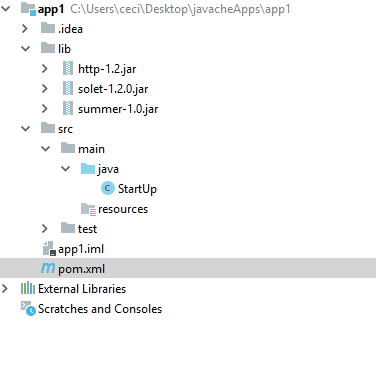
So to get started, you need to know how **Broccolina** works. Broccolina extracts every .jar file that was present in the webapps folder during the initial start of the server. Then scans the extracted folder for .class files and filters those, who are assignable from BaseHttpSolet. Then those files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

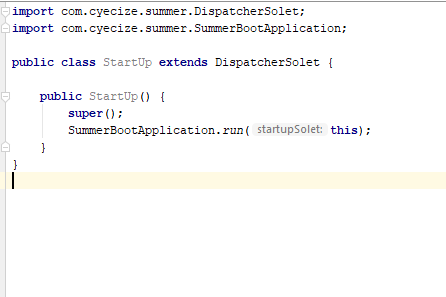


We want to create a folder named lib where we will include the 3 mandatory dependencies for out app to compile and run properly. Those are Http, SoletApi and Summer MVC.

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);



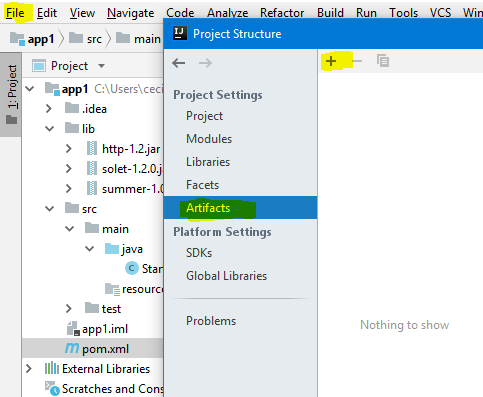
That’s it, now your application will be recognized by Broccolina. But to run the app we also need to package It in the proper way, which is:

classes folder what contains the compile output.

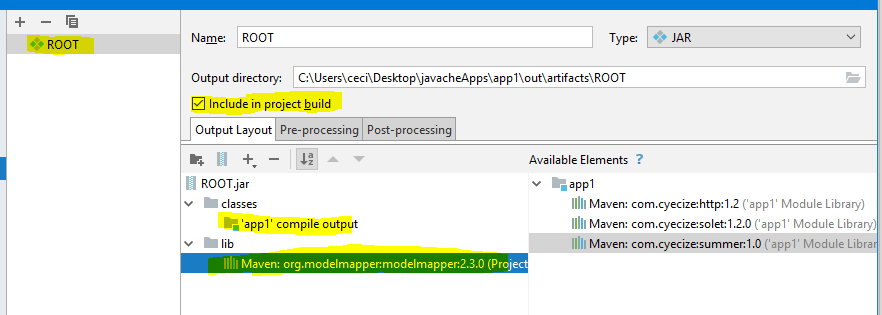
lib folder to place your third party libraries.

**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

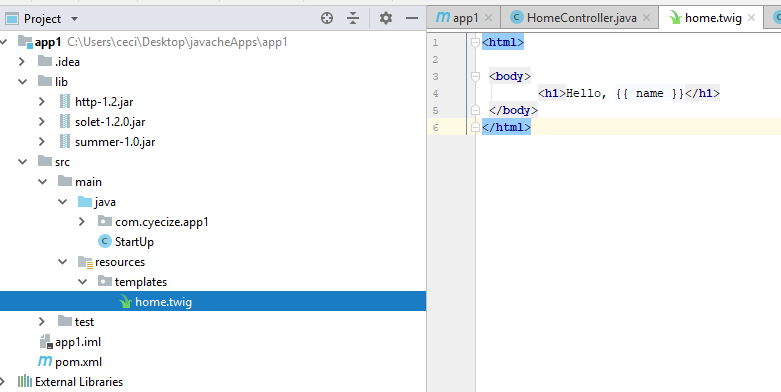
Summer MVC outside because they will be in the javache’s lib folder.

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

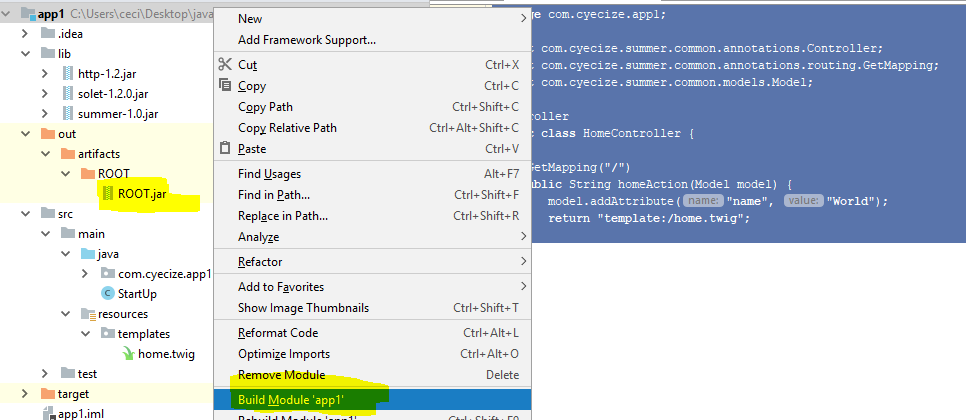
Create a controller HomeController, annotate it with @Controller and add a public String homeAction, annotated with @GetMapping.

**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"**;  
 }  
}

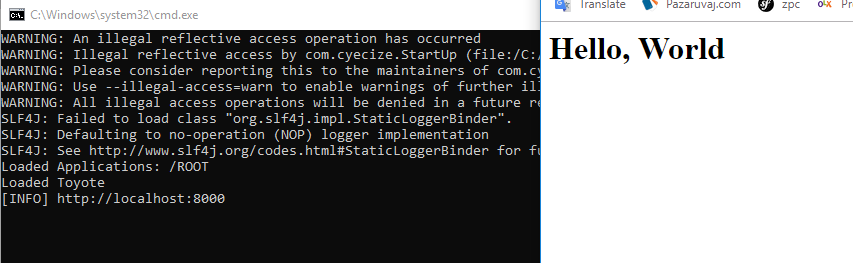
After that in your resources folder, add new folder and name it templates. Inside create a file named home.twig.



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



And then run the app.



**Voilà!**

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.

Routing

**@GetMapping, @PostMapping** are the annotations that are used for initializing a route.

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

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.

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

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.

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.