Spring MVC Framework

Das Spring MVC Framework implementiert die Idee Model View Controller. Wir erreichen dadurch eine gute Trennung der verschiedenen Aspekte einer Web-Applikation.

Die drei Teile «input logic», «business logic» und «UI logic» können so unabhängig voneinander erstellt und gepflegt werden.

Controller

Der Controller nimmt Benutzeranfragen entgegen, löst Aktionen im Service-Layer aus, sammelt die Resultate und bestimmt, welche View ausgeliefert werden soll.

Model

Das Model besteht aus Business Objekten welche typischerweise als POJOs (plain old Java objects) implementiert sind.

View

Die ist verantwortlich für die Darstellung der Modeldaten und liefert im Allgemeinen HTML-Dateien aus.

Dreh- und Angelpunkt ist dabei der Controller. Sie können sich das folgendermassen vorstellen:

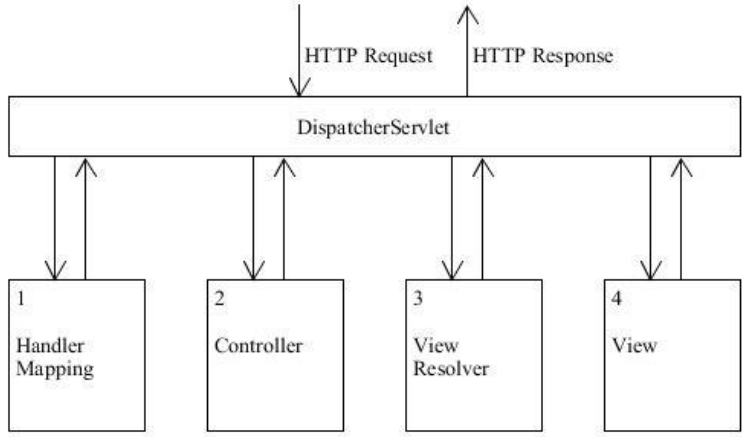

Abbildung 1 Spring MVC

Ein zentrales «DispatcherServlet» nimmt Anfragen entgegen, bestimmt mit Hilfe des «HandlerMappings», welcher Controller, und dort, welche Funktion zuständig ist für die Bearbeitung der Anfrage. Anschliessend wird die Funktion ausgeführt. Der Controller macht seinen Job und gibt anschliessend den Namen der View zurück, welche angezeigt werden soll. Der «ViewResolver» sucht diese View und das «DispatchServlet» übergibt der View die Daten, welche der Controller besorgt hat. Die View wird passend gerendert und anschliessend ausgeliefert.

Eine detailliertere Übersicht sehen Sie unter Spring MVC (http://terasolunaorg.github.io/guideline/1.0.1.RELEASE/en/Overview/SpringMVCOverview.html#id1)

Spring Annotations für Controller

Wir werden folgende Annotations für unsere Controllerklassen brauchen:

@Controller

Bestimmt, dass diese Klasse von Spring MVC als Controller erkannt wird

@RequestMapping(URL)

Bezeichnet eine Funktion als Zuständig für diese URL. Das Format der URL kann einfach sein (z.B. «/home») oder es kann Platzhalter enthalten (z.B. «/user/showCustomer/{id}»).

@PathVariable(«id»)

Definiert, dass die Nachfolgende Variable mit einem Teil der «RequestMapping-URL» (dem Platzhalter «id») initialisiert werden soll.

@RequestParam(«paramName»)

Definiert, dass die Nachfolgende Variable initialisiert wird mit dem Wert, welcher unter dem Namen «paramName» aus dem Formular übertragen wurde.

Bestimmen der auszuliefernden View

Jede Controllerfunktion gibt einen String zurück, welche die auszuführende Aktion bezeichnet. Beginnt der String mit «redirect:» gefolgt von einer URL, wird eine neue Anfrage an das «DispatcherServlet» gesendet. Andernfalls bezeichnet der String den Namen der View, welche ausgeliefert werden soll.

Übergabe von Daten an die View

Sollen Daten an die View weitergegeben werden, braucht die Controllerfunktion einen Parameter vom Typ Model. Wir können unsere Werte in diesem Model mittels der Funktion «addAttribute(<name>,< value>)» übergeben. In der View können wir dann unter «<name>» auf den übergebenen Wert «(<value>)» zugreifen.

Beispiel «CustomerController»

Der untenstehende Code zeigt alle vorgängig besprochenen Features:

```
/**
 * Customer Web Controller </br>
 * maps any customer related URL and processes them
 */
@Controller
public class CustomerController {

	@Autowired
	private CustomerService customerService;

	@RequestMapping("/")
	public String root() {
		return "redirect:home";
	}

	@RequestMapping("/home")
	public String showCustomerList(Model model) {
		List<Customer> customers = customerService.getCustomerList();
		model.addAttribute("customers", customers);
		return "showCustomerList";
	}

	@RequestMapping("/user/showCustomer/{id}")
	public String showCustomer(
	  Model model, @PathVariable("id") long customerId) {
		model.addAttribute("customer",customerService.getCustomer(customerId));
		return "showCustomer";
	}

	@RequestMapping("/user/addMemo")
	public String addMemo(Model model,
			@RequestParam("customerId") long customerId,
			@RequestParam("memoText") String memoText)
	{
		customerService.addMemoToCustomer(customerId, memoText);
		return "redirect:/user/showCustomer/" + customerId;
	}
}
```