Spring Web MVC

MVC - podsećanje

- MVC = Model - View - Controller
- Model => Java klase koje predstavljaju model podataka nad kojim aplikacija radi
- View => komponente aplikacije zadužene za vizuelizaciju (html bazirani templejti, frontend), tj. formatiranje odgovora za klijenta
- Controller => upravlja tokom podataka i izmenama u modelu, a osvežava View nastalim promenama

Spring Web MVC

- Spring Web MVC je originalni okvir za razvoj web aplikacija baziran na **Servlet API** specifikaciji

- Postoji kao deo Spring Framework-a od samog početka.

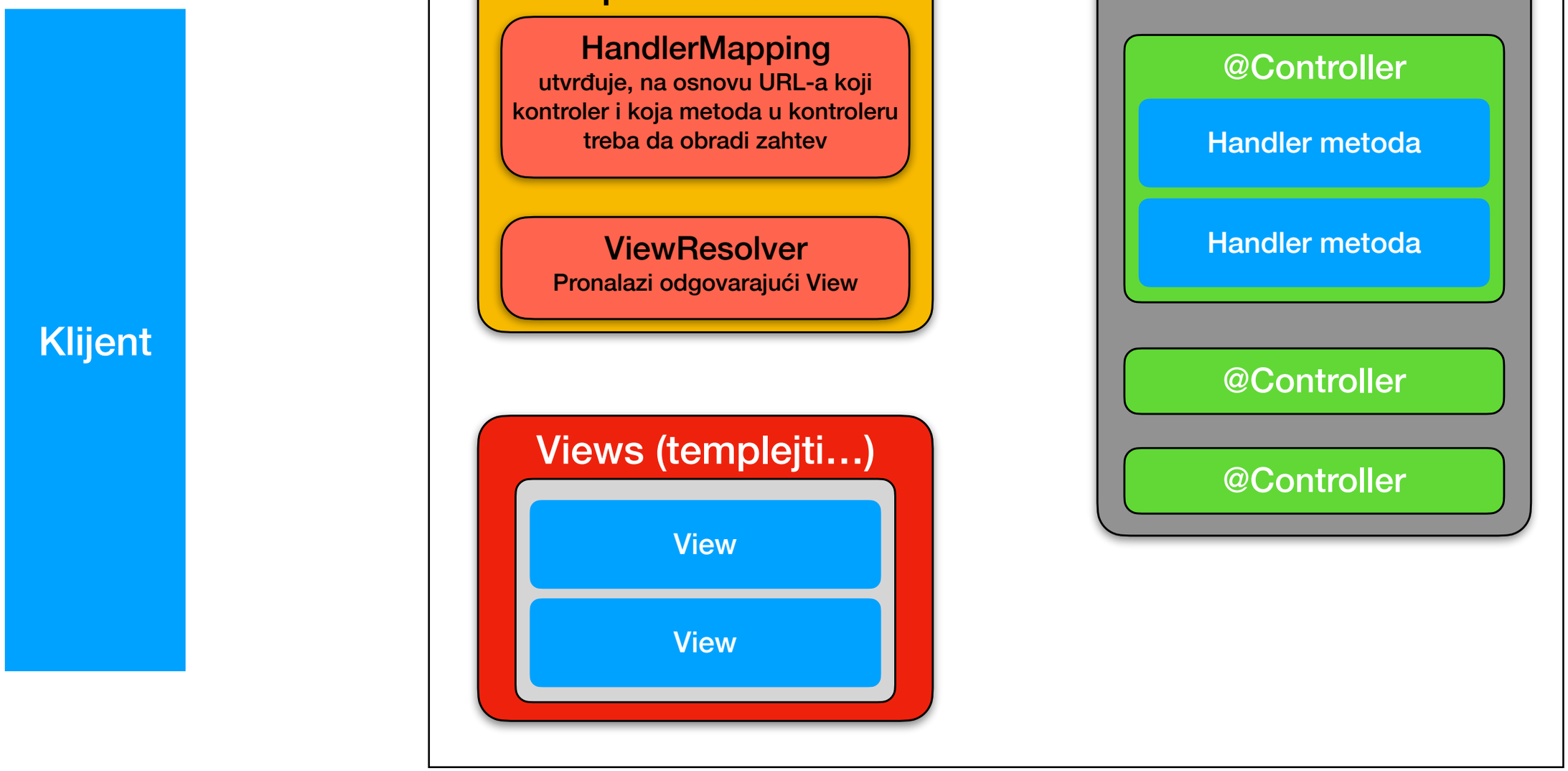
- Sam naziv, “Spring Web MVC,” je istovremeno i naziv osnovnog modula (spring-webmvc), ali se češće govori samo o “Spring MVC”.

Spring Web MVC DispatcherServlet

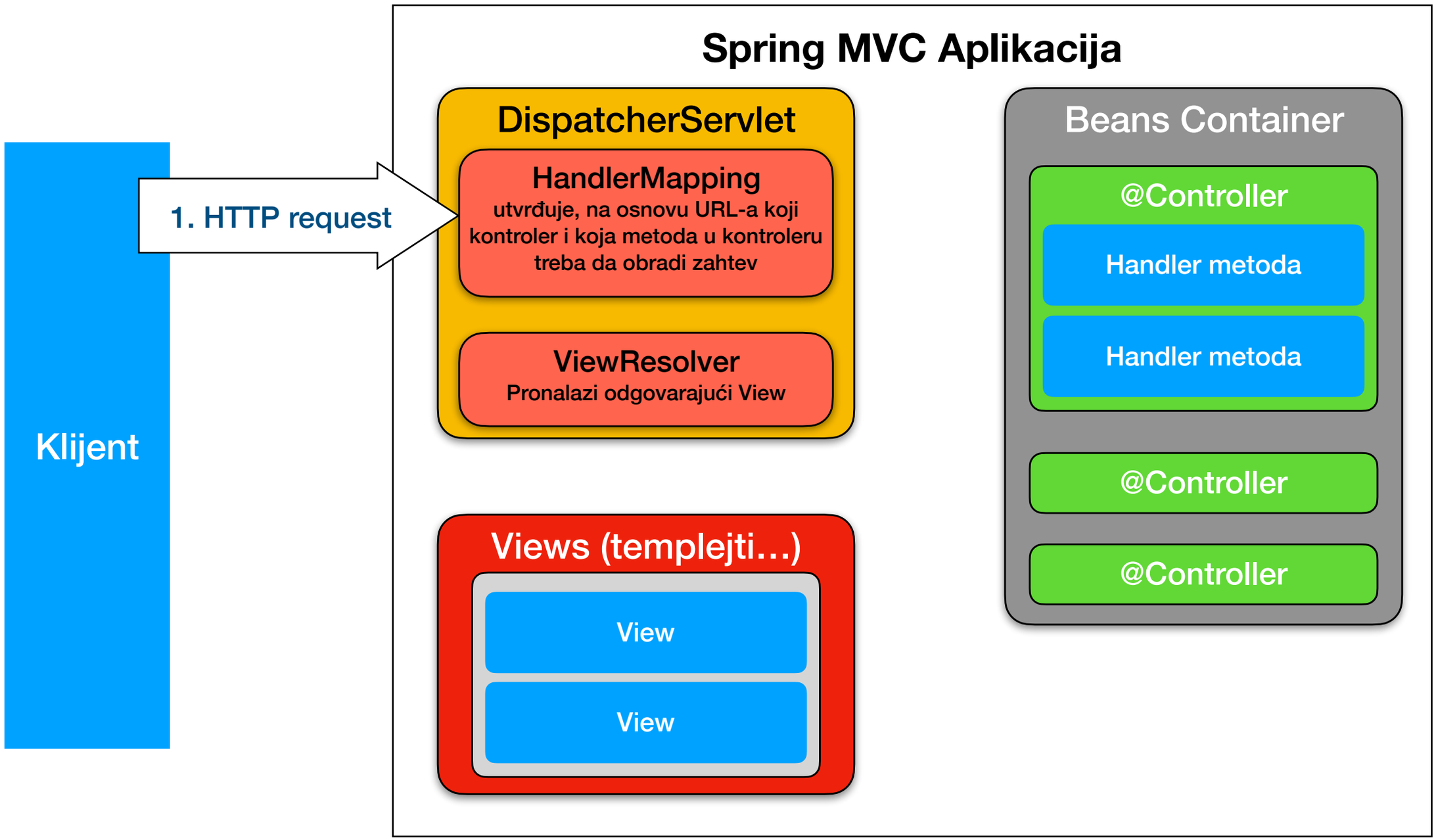
- Spring Web MVC kao i mnogi drugi MVC web okviri za razvoj (frameworks), zasniva se na tzv. FrontController obrazac (šablonu). Front Controller predstavlja centralno mesto koje prima zahtev korisnika i na osnovu sadžaja samog zahteva odlučuje kojoj komponenti treba proslediti zahtev na dalju obradu.

- U Spring MVC, ulogu front controllera obavlja **DispatcherServlet**, koji sadrži logiku za analizu sadržaja zahteva i utvrđivanje koja komponenta je zadužena za njegovo dalje procesiranje.

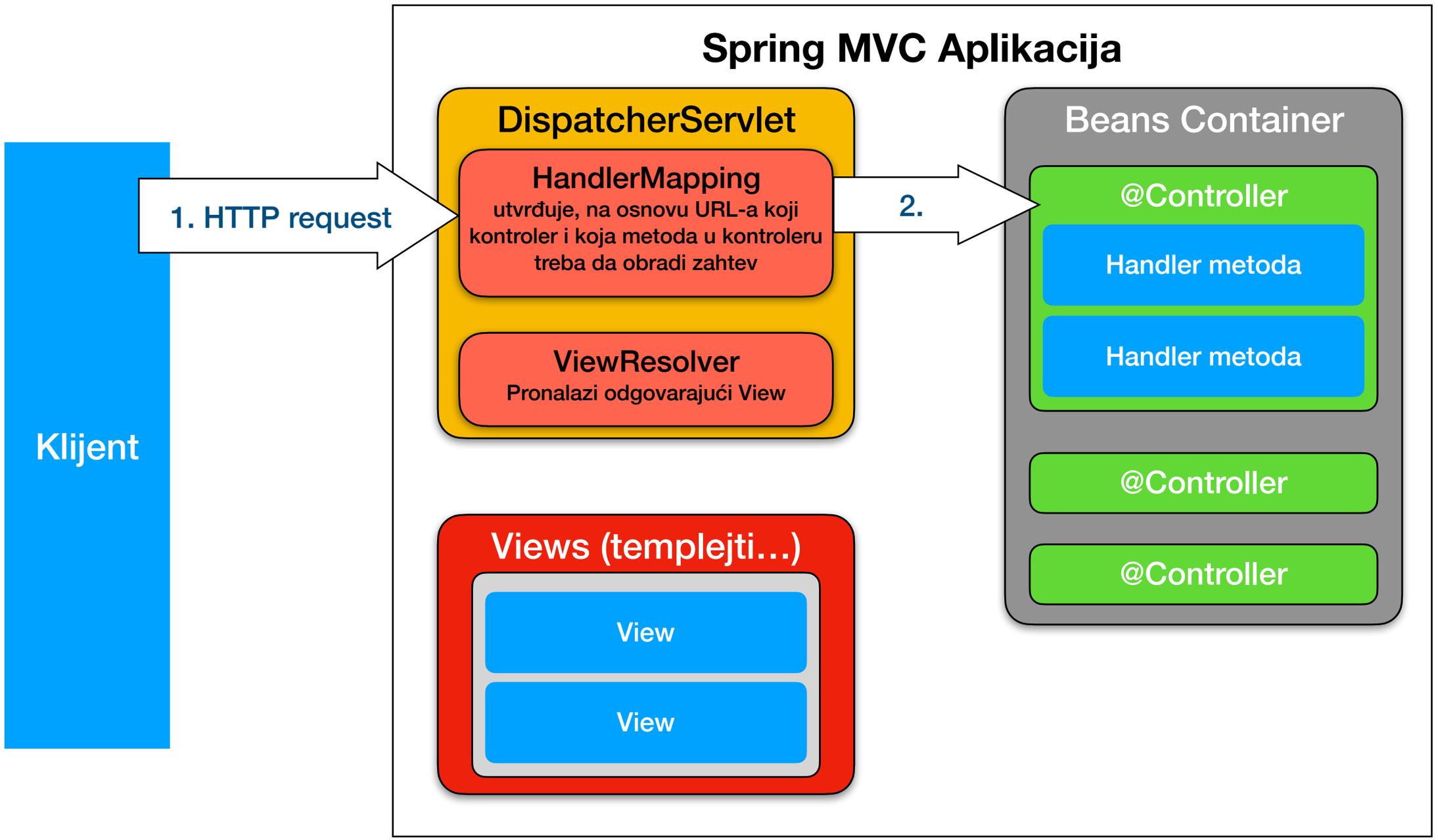
Spring Web MVC
tok obrade zahteva



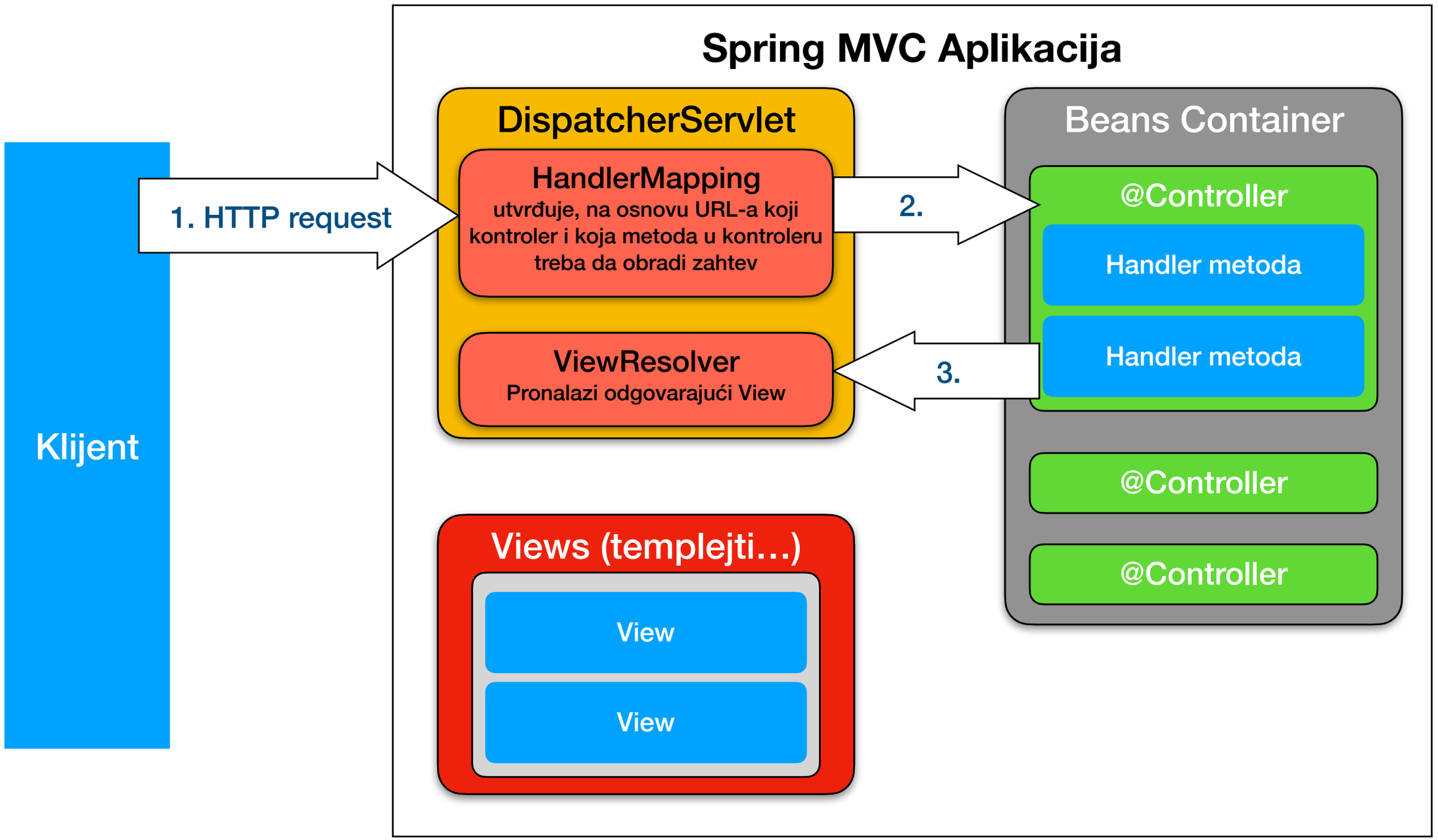
Spring Web MVC
tok obrade zahteva



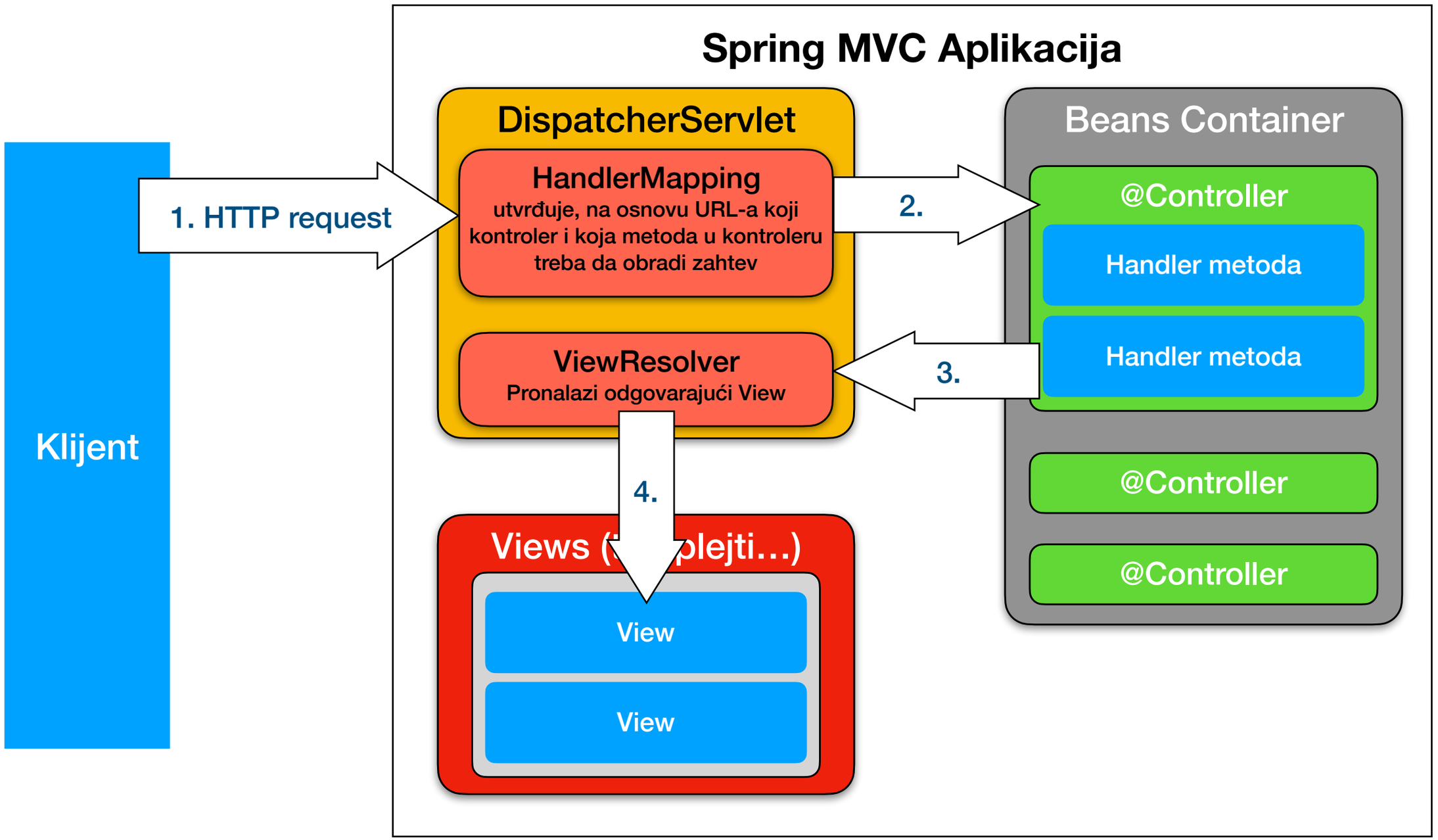
Spring Web MVC
tok obrade zahteva



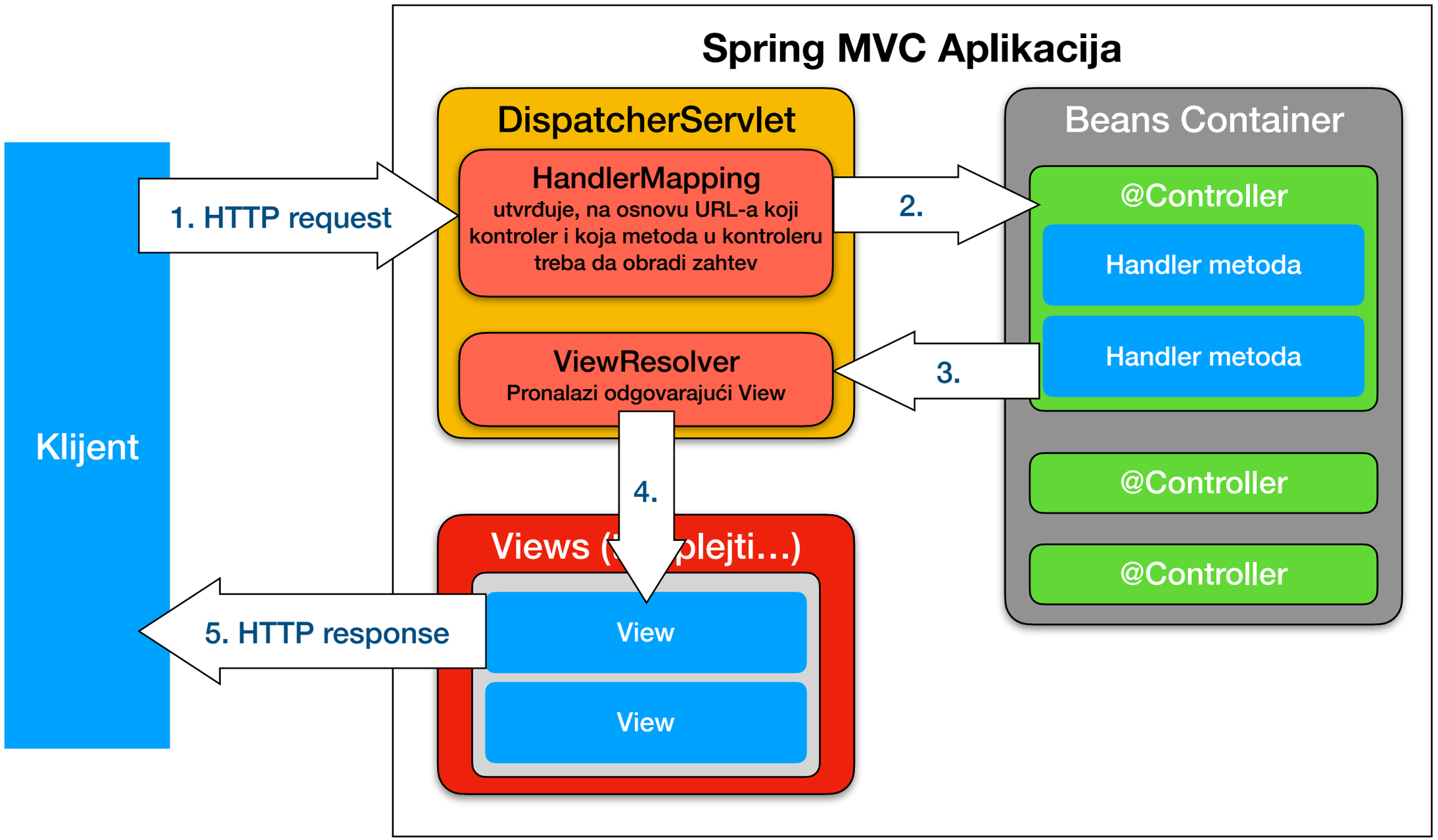
Spring Web MVC
tok obrade zahteva



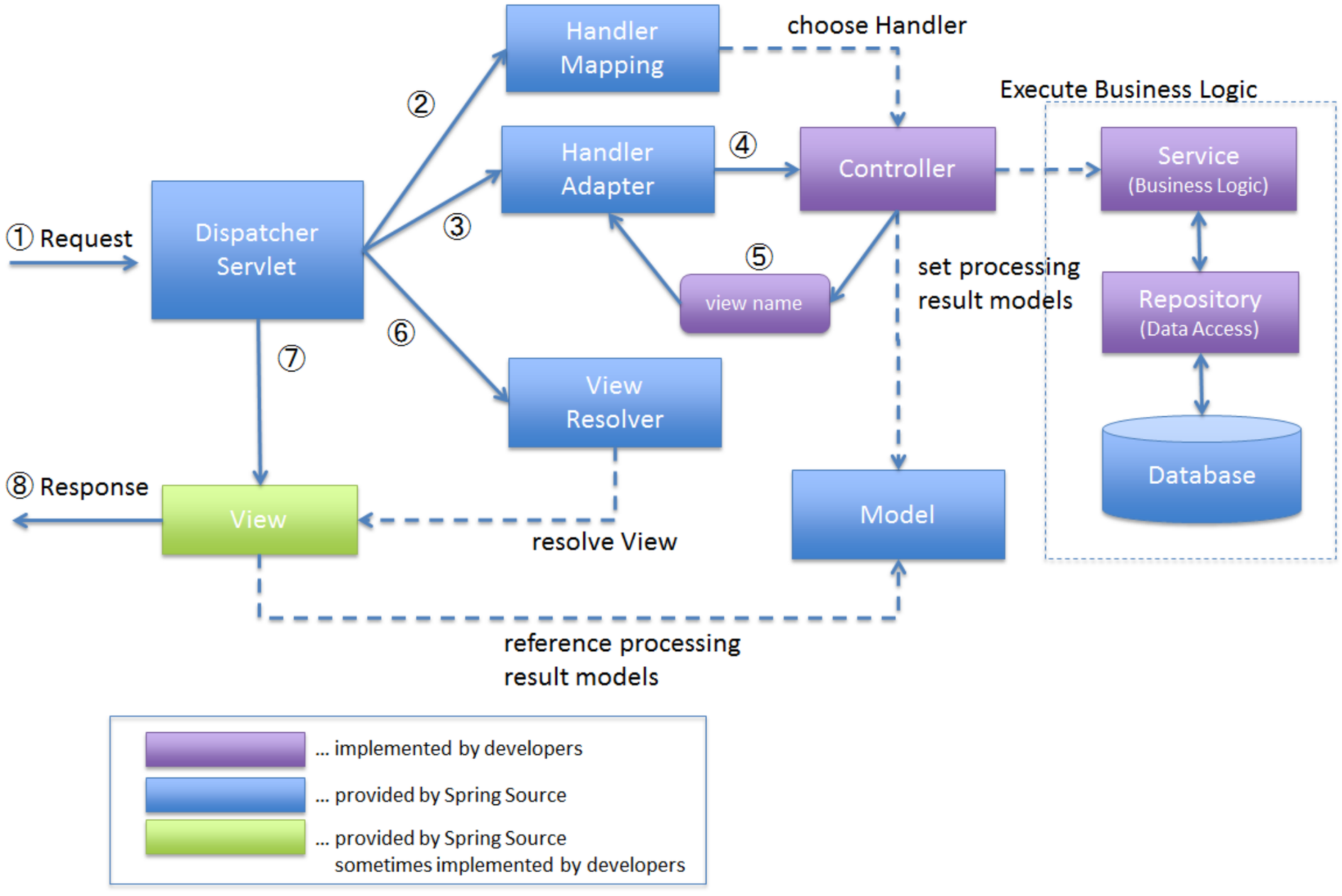
Spring Web MVC
tok obrade zahteva



Spring Web MVC
tok obrade zahteva



Spring Web MVC
tok obrade zahteva - detaljnije



Slika preuzeta sa: https://terasolunaorg.github.io/guideline/5.0.x/en/Overview/SpringMVCOverview.html

Spring Web MVC
tok obrade zahteva - koraci

1. DispatcherServlet prima zahtev.
2. DispatcherServlet prosleđuje zadatak određivanja odgovarajućeg kontorlera HandlerMapping komponenti. HandlerMapping bira odgovarajući kontroler - onaj koji je napamiran na URL na koji je klijentski zahtev upućen. Vraća DispathcerServletu informaciju o Controlleru i Handleru kome se zahtev treba proslediti.
3. DispatcherServlet delegira prosleđivanje zahteva HandlerAdapter komponenti, koja prilagođava sadržaj zahteva za poziv metode kontrolera na odgovarajući način.
4. HandlerAdapter poziva odgovarajuću metodu kontrolera Controller.
5. Controller poziva poslovnu logiku (Service komponente) iz svoje metodi, i postavlja odgovarajuće rezultate u Model objekat, a osim toga treba da HandlerAdapteru vrati i iinformaciju koju View komponentu treba iskoristiti za prosleđivanje podataka iz modela ka korisniku.
6. DispatcherServlet prosleđuje zadatak pronalaženja odgovarajuće View komponente ViewResolver-u. ViewResolver vraća View kompnentu koja je mapirana na naziv View-a koji joj je bio prosleđen.
7. DispatcherServlet poziva proces “renderinga” vraćene View komponente.
8. View izvršava rendering i vraća reaultat klijentu. (Rendering proizvodi konačni sadržaj koji se vraća klijentu npr. popunjava HTML templejt podacima iz Modela i proizvodi gotov HTML)

Konfigurisanje DispatcherServleta

- Kao i svaki drugi servlet i DispatcherServlet mora da se konfiguriše (defaultna konfiguracija postoji pa najčešće nije potrebna posebna).
- Naknadno koristeći Spring configuration “otkriva” komponente kojima delegira reqqest mapping, view resolution, exception handling…

```
public class MyWebApplicationInitializer implements WebApplicationInitializer {
    @Override
    public void onStartup(ServletContext servletCxt) {
        // Load Spring web application configuration
        AnnotationConfigWebApplicationContext ac =
            new AnnotationConfigWebApplicationContext();
        ac.register(AppConfig.class);
        ac.refresh();
        // Create and register the DispatcherServlet
        DispatcherServlet servlet = new DispatcherServlet(ac);
        ServletRegistration.Dynamic registration =
            servletCxt.addServlet("app", servlet);
        registration.setLoadOnStartup(1);
        registration.addMapping("/app/*");
    }
}
```

Spring Web MVC

- U nastavku ćemo detaljnije analizirati ulogu pojedinih komponenti *framework-a* i anotacije koje se koriste da bi se određenim klasama dodelila uloga komponenti u Spring MVC frameworku.

Spring MVC - Controllers

@Controller

- Ova anotacija označava da data klasa ima ulogu controllera u MVC

- Controller-i su komponente za prihvatanje i obradu klijentskih zahtreva (HTTP requests)

- Tipično se unutar klase nalaze *handling* metode anotirane pomoću @RequestMapping anotacije kako bi se obezbedilo mapiranje URL-a na dati metod

@Controller

```
@Controller
public class MyController {

    @RequestMapping(value = "/getDateAndTime")
    public ModelAndView getDateAndTime() {

        var now = LocalDateTime.now();
        var dtf = DateTimeFormatter.ofPattern(
                        “dd.MM.yyyy HH:mm:ss");
        var date_time = dtf.format(now);

        var data = new HashMap<String, Object>();
        data.put("date_time", date_time);

        return new ModelAndView("showMessage", data);
    }
}
```

@Controller

```
@Controller
public class MyController {...}
```

- Ova anotacija označava da data klasa MyController ima ulogu kontrolera za našu MVC baziranu web aplikaciju

@RequestMapping

- **@RequestMapping** anotacija može da se koristi kako na samu klasu tako i na metode klase

- Mapira određeni URL na celu klasu ili njene metode

- Ukoliko se anotacija iskoristi ispred klase - onda je to bazni url koji klasa obrađuje, a u metodama se može dodati specifično procesiranje url-a

@RequestMapping

```
@RequestMapping(value = "/getDateAndTime")
public ModelAndView getDateAndTime(){...}
```

- metoda **getDateAndTime** je *handler* za procesiranje bilo kog requesta koji je upućen ka URL-u: **http://<hostname>/<appbaseURL>/getDateAndTime**

- U ovom slučaju vraća objekat klase **ModelAndView**

Šta je ModelAndView?

- Objekat ove klase služi kao “nosač” za Model i View objekte. Oni ostaju potpuno odvojeni, ova klasa sadrži samo referencu na odgovarajući objekat koji predstavlja model i odgovarajuću View komponentu koja treba da isprocesira prikaz (fromatiranje podataka) klijentu.

- Objekat ove klase samo omogućava da kontroler vrati obe komponente u jednom “nosećem” objektu.

- Reprezentuje model podataka koji treba iskoristiti tokom popunjavanja prikaza za vreme renderinga, kao i *view* koji za to treba iskoristiti.

- Informacija o tome koji *view* treba koristiti se može proslediti kao String koji se onda mora rezolvirati kroz ViewResolver, i alternativno može se proslediti direktno View objekat.

Šta je ModelAndView?

```
var data = new HashMap<String, Object>();
data.put("date_time", date_time);

return new ModelAndView("showMessage", data);
```

- **data** - model podataka - predstavlja **mapu** objekata koji se prosleđuju prikazu (view) za vizuelizaciju

- **showMessage** - String koji predstavlja **naziv** view komponente koju treba iskoristiti za prikaz podataka (najčešće neki HTML templejt npr. Freemarker ili Thymeleaf template)

@RequestMapping koji “hvata” više url-ova

```
@Controller
@RequestMapping(“welcome")
public class HomeController {

  @RequestMapping(value={"", "/index", "page*","view/*"})
  String welcomePage(){
    return "Hello from index multiple mapping.";
  }
}
```

- Ako nam je aplikacija namapirana na http://localhost:8080 bazni url koji “gađa” ovu klasu je
  http://localhost:8080**/welcome**

- ali ovakvo mapiranje obezbeđuje i da se ova metoda pozove da procesira request koji stigne na bilo koji od sledećih url-ova:
  http://localhost:8080/**welcome**, http://localhost:8080/**welcome/index**,
  http://localhost:8080/**welcome/page**, http://localhost:8080/**welcome/page123**,
  http://localhost:8080/**welcome/view/**, http://localhost:8080/**welcome/view/something**

@RequestMapping za različite HTTP metode

```
@Controller
@RequestMapping("/home")
public class IndexController {
  @RequestMapping(method = RequestMethod.GET)
  String get(){
    return "Hello from get";
  }
  @RequestMapping(method = RequestMethod.DELETE)
  String delete(){
    return "Hello from delete";
  }
  @RequestMapping(method = RequestMethod.POST)
  String post(){
    return "Hello from post";
  }
  @RequestMapping(method = RequestMethod.PUT)
  String put(){
    return "Hello from put";
  }
  @RequestMapping(method = RequestMethod.PATCH)
  String patch(){
    return "Hello from patch";
  }
}
```

@RequestMapping za različite HTTP metode

- Verzije Spring frameworka posle 4.3 uveli su skraćene anotacije za mapiranje na pojedine metode

umesto:

```
@RequestMapping(value = "/get/{id}", method = RequestMethod.GET)
```

skraćenom anotacijom moguće je pisati:

```
@GetMapping("/get/{id}")
```

@RequestMapping za različite HTTP metode

- Skraćena anotacija za HTTP metode
  - *@GetMapping*
  - *@PostMapping*
  - *@PutMapping*
  - *@DeleteMapping*
  - *@PatchMapping*

@RequestParam

- @RequestParam se koristi za ekstrahovanje parametara upita, parametara iz formi i fajlova iz zahteva.

```
@GetMapping("/prod")
public String getProd(@RequestParam String id) {
    return "ID: " + id;
}
```

- U ovom primeru, iz poslanog zahteva se ekstrahuje **id** parametar http://localhost:8080/prod?**id=PID12345**

@RequestParam - name value

- @RequestParam anotacija može biti dodatno opisana atributima ***name, value, required i defaultValue***

- Ukoliko se ***name*** ne navede podrazumeva se da je isti kao naziv varijable, ali ako ga navedemo onda tražimo parametar po imenu i vrednost smeštamo u varijablu koja je iza anotacije

```
@PostMapping("/prod")
public String addProd(@RequestParam(name=“id”) String prodID,
        @RequestParam  String pName){
    return "ID: " + prodID + “Product name: “+pName;
}
```

- http://localhost:8080/prod?id=PID12345&pName=Plazma

- može se koristiti i alternativni value atribut ili skraćena anotacija @RequestParam(value = “id”) ili samo @RequestParam(“id”).

@RequestParam - required

- @RequestParam može biti i opcioni ukoliko navedemo da je required=false
- U tom slučaju treba voditi računa da taj parametar može imati i null vrednost ukoliko nije prosleđen

```
@GetMapping("/prod")
public String getProd(@RequestParam(required=false)
                      String id) {
    return "ID: " + id;
}
```

@RequestParam - defaultValue

- @RequestParam može biti i opcioni ukoliko navedemo da je required=false
- U tom slučaju treba voditi računa da taj parametar može imati i null vrednost ukoliko nije prosleđen

```
@GetMapping("/prod")
public String getProd(@RequestParam(defaultValue=“P123”)
                      String id) {
    return "ID: " + id;
}
```

@RequestParam - mapiranje svih parametara

- Parametri zahteva su dostupni i kao map tipa Map<String, String>, pa je moguće odjednom preuzeti sve, bez potrebe da se navode jedan po jedan.

```
@PutMapping("/prod")
public String updateProd(@RequestParam Map<String,String>
                                allParameters) {
    return “all parameters: " + allParameters.entrySet();
}
```

@PathVariable

- Omogućava da se iz URL-a ekstrahuju podaci koji su namapirani na tzv. pozicione parametre (nisu u klasičnom query stringu).

```
@GetMapping(“/prod/{id}“)
public String getProd(@PathVariable String id) {
    return "ID: " + id;
}
```

- Path variable nisu URL enkodirane za razliku od Request parametara

- http://localhost:8080/prod/PID12345

@RequestBody

- Omogućava automatsko mapiranje sadržaja tela poruke na domenski objekat ili transfer objekat (**DTO**)

- Automatski deserijalizuje JSON u Java objekat.
  Java tip koji je naveden posle **@RequestBody** mora odgovarati sadržaju JSON zapisa

```
@PostMapping("/request")
public ResponseEntity postController(
  @RequestBody LoginForm loginForm) {

    exampleService.fakeAuthenticate(loginForm);
    return ResponseEntity.ok(HttpStatus.OK);
}
```

@ResponseBody

- ova anotacija saopštava kontroleru da sadržaj odgovora treba automatski da se serijalizuje u određenu notaciju

```
@Controller
@RequestMapping("/post")
public class ExamplePostController {

    @Autowired
    ExampleService exampleService;

    @PostMapping(“/response",produces = MediaType.APPLICATION_JSON_VALUE)
    @ResponseBody
    public ResponseTransfer postResponseController(
      @RequestBody LoginForm loginForm) {
        return new ResponseTransfer("Thanks For Posting!!!");
     }
}
```

@ResponseBody

- Isti kontroler može imati namapirane različite metode za isti tip zahteva, koje produkuju različiti response

```
@PostMapping(value=“/content",produces=MediaType.APPLICATION_JSON_VALUE)
@ResponseBody
public ResponseTransfer postResponseJsonContent(
  @RequestBody LoginForm loginForm) {
    return new ResponseTransfer("JSON Content!");
}

@PostMapping(value="/content",produces = MediaType.APPLICATION_XML_VALUE)
@ResponseBody
public ResponseTransfer postResponseXmlContent(
  @RequestBody LoginForm loginForm) {
    return new ResponseTransfer("XML Content!");
}
```

@RestController

- anotacija koja govori MVC frameworku da se radi o kontroleru koji funkcioniše kao endpoint RESTful servisa

- Primarna namena je pojednostavljene zapisa, jer ova anotacija automatski uključuje i @Controller i @ResponseBody anotacije

- rest kontroler jeste kontroler

- rezultat svake handler funkcije treba da se vrati kao JSON zapis u telu poruke

Interface Response<T>

- reprezentuje tipizirani HTTPResponse koji se vraća kao rezultat handler metode

Class ResponseEntity<T>

- ResponseEntity objedinjuje sadržaj celokupnog HTTP odgovora u jedan objekat (status kod, header-e, body)
- Pomoću njega se direktno na jednom mestu može u potpunosti konfigurisati HTTP Response
- Sadrži dva statička interfejsa:
  - BodyBuilder
  - HeaderBuilder

Class ResponseEntity<T>

- Olakšava postavljanje željenih podešavanja delova HTTP reponsa

```
@GetMapping("/customHeader")
ResponseEntity<String> customHeader() {
    return ResponseEntity.ok()
        .header("Custom-Header", "foo")
        .body(“Custom header set");
}
```

- Za jednostavnije stvari možda je lakše koristiti neku od alternativa

Alternative za ResponseEntity<T>

- Upotreba @ResponseBody anotacije

- Upotreba @ResponseStatus anotacije (može i u kombinaciji sa @ExceptionHandler)

```
@ResponseStatus(HttpStatus.I_AM_A_TEAPOT)
void teaPot() {}
. . .
@ResponseStatus(HttpStatus.BAD_REQUEST,
                reason = "Some parameters are invalid")
void onIllegalArgumentException(IllegalArgumentException
exception) {}
```

- Upotreba @ResponseStatus anotacije u kombinaciji sa @ExceptionHandler

Alternative za ResponseEntity<T>

- Direktna manipulacija HTTPServletResponse objekta

```
@GetMapping("/manual")
void manual(HttpServletResponse response) throws IOException
{
    response.setHeader("Custom-Header", "foo");
    response.setStatus(200);
    response.getWriter().println("Hello World!");
}
```