Spring (REST) Services

Philippe Collet

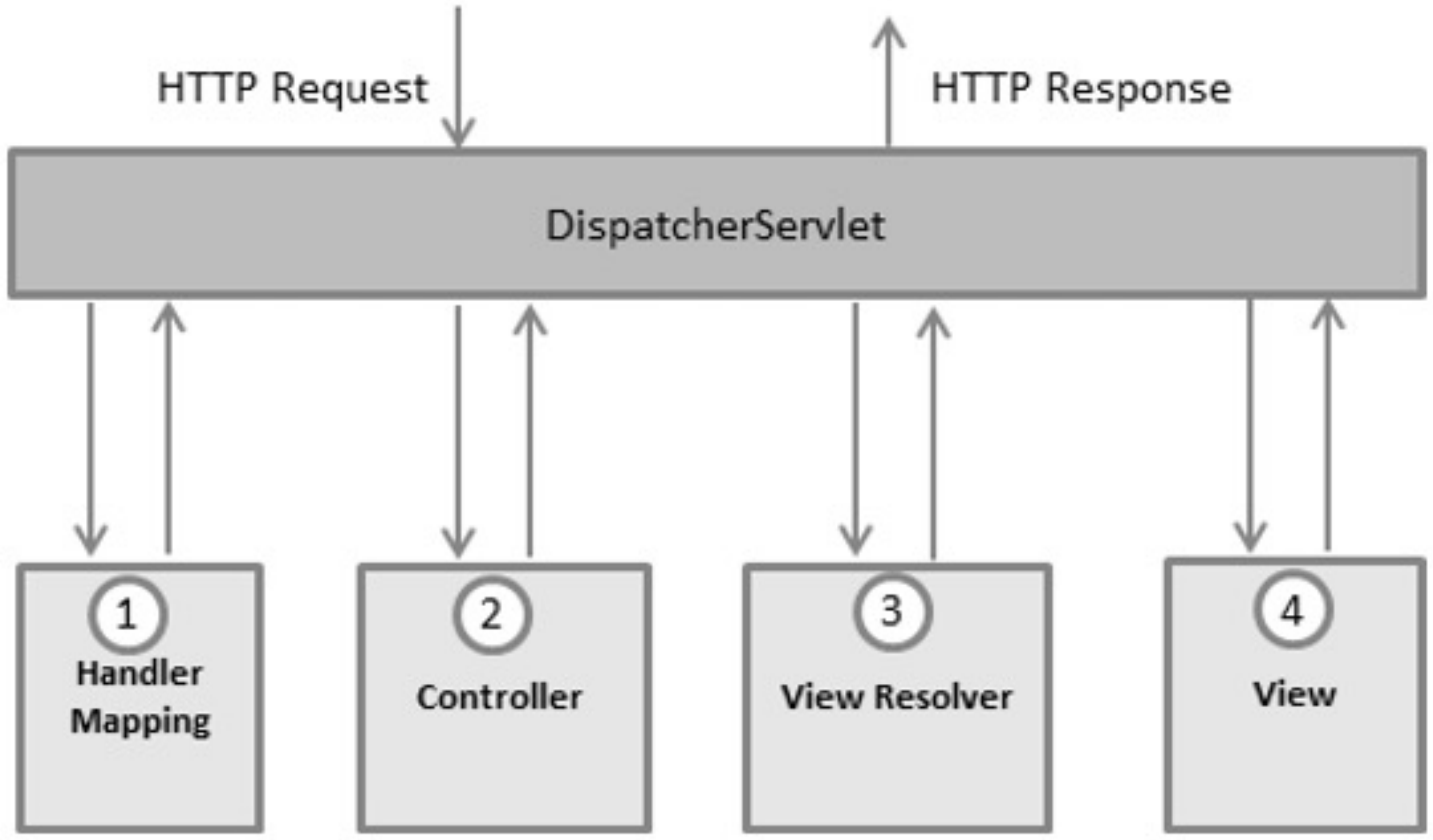
Everything started with SpringMVC

- Aim: building flexible and loosely coupled web applications

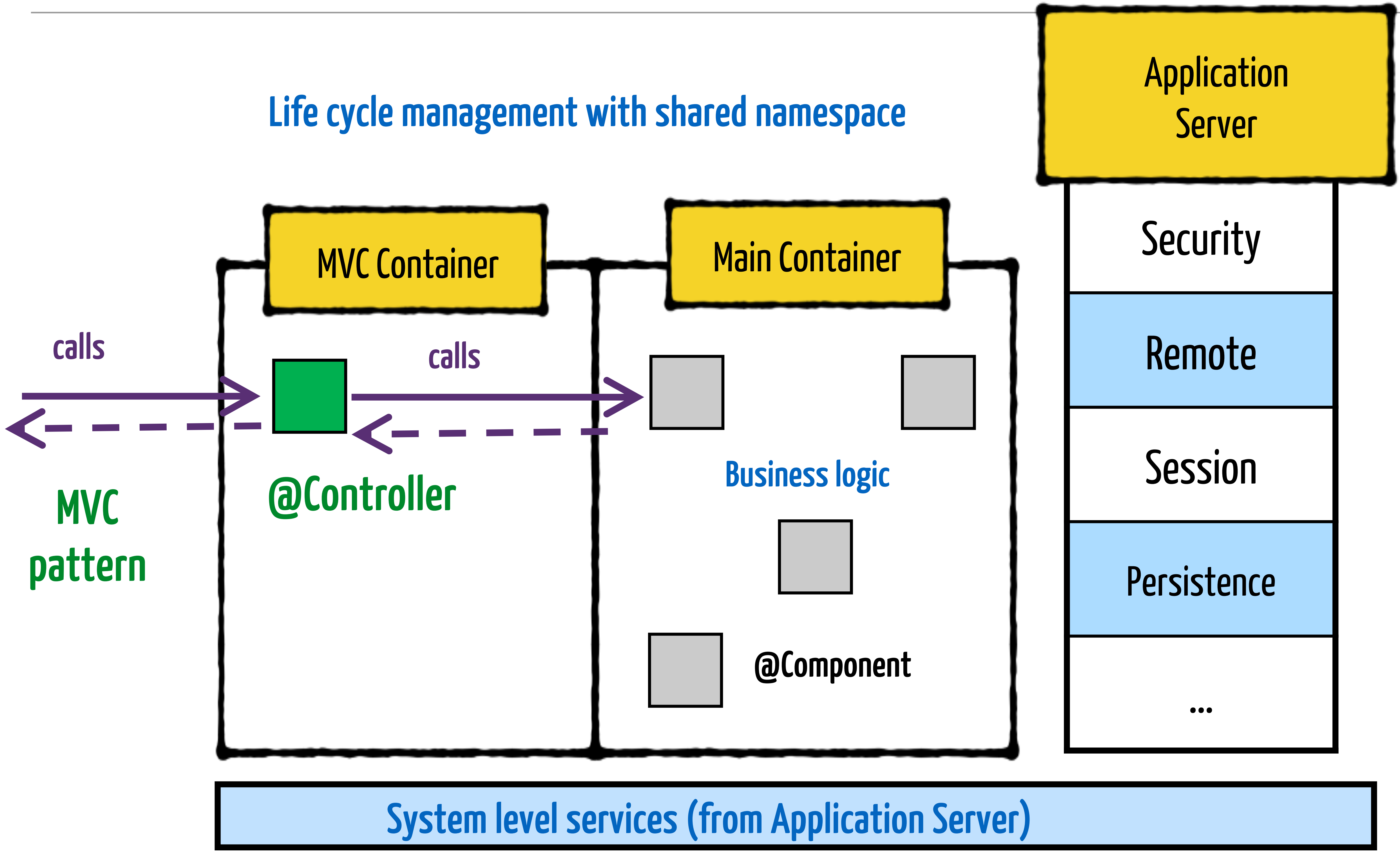
- Model-view-controller design pattern helps in separating
    - the business logic,
    - presentation logic,
    - navigation logic.

- Models are responsible for encapsulating the application data (POJOs)
- The Views render response to the user with the help of the model object (HTML)
- Controllers are responsible for receiving the request from the user, calling the back-end services, and passing the resulting model to the right view

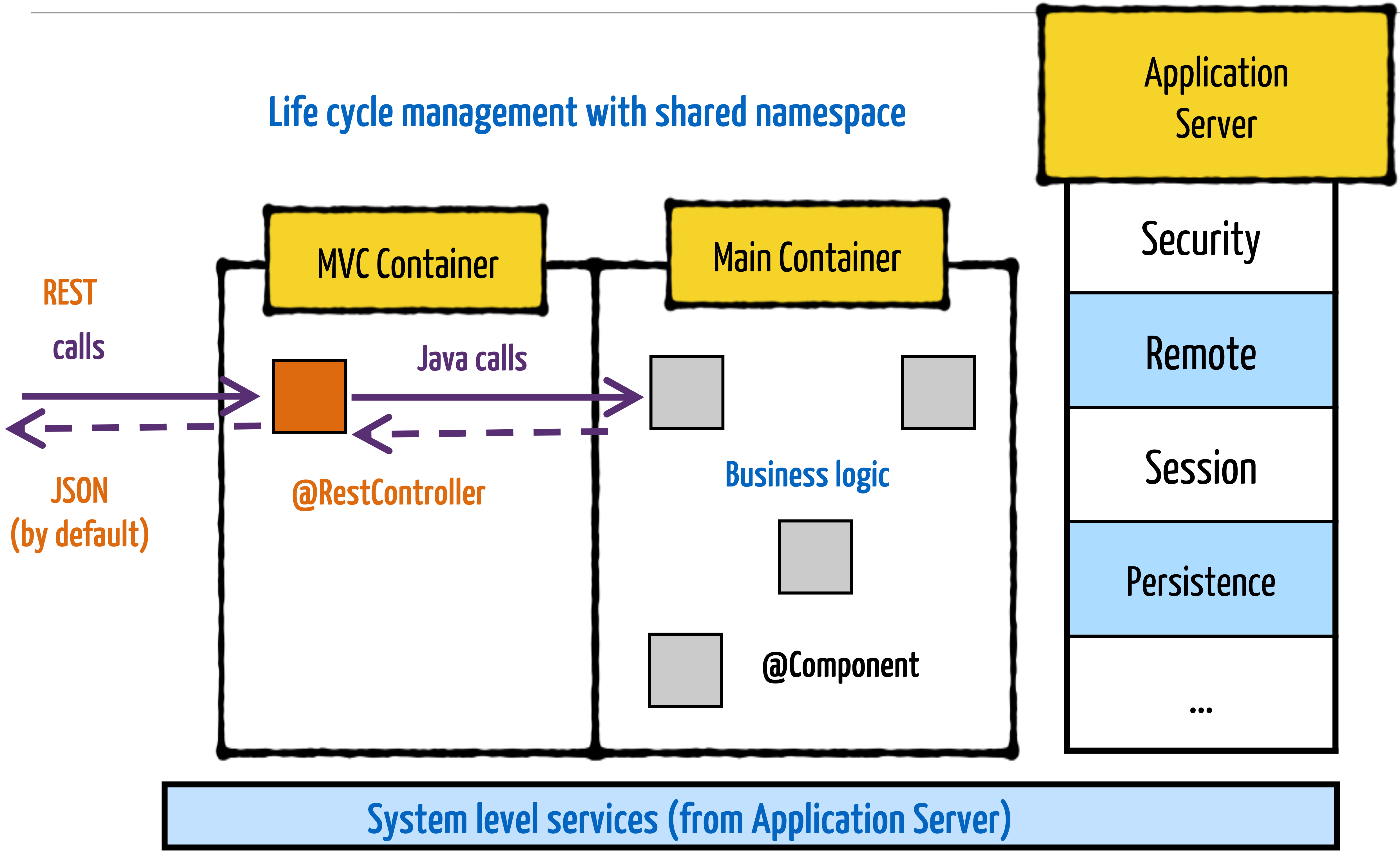
Spring MVC architecture and behavior



Spring MVC infrastructure



Spring MVC infrastructure for REST



Spring WebFlux

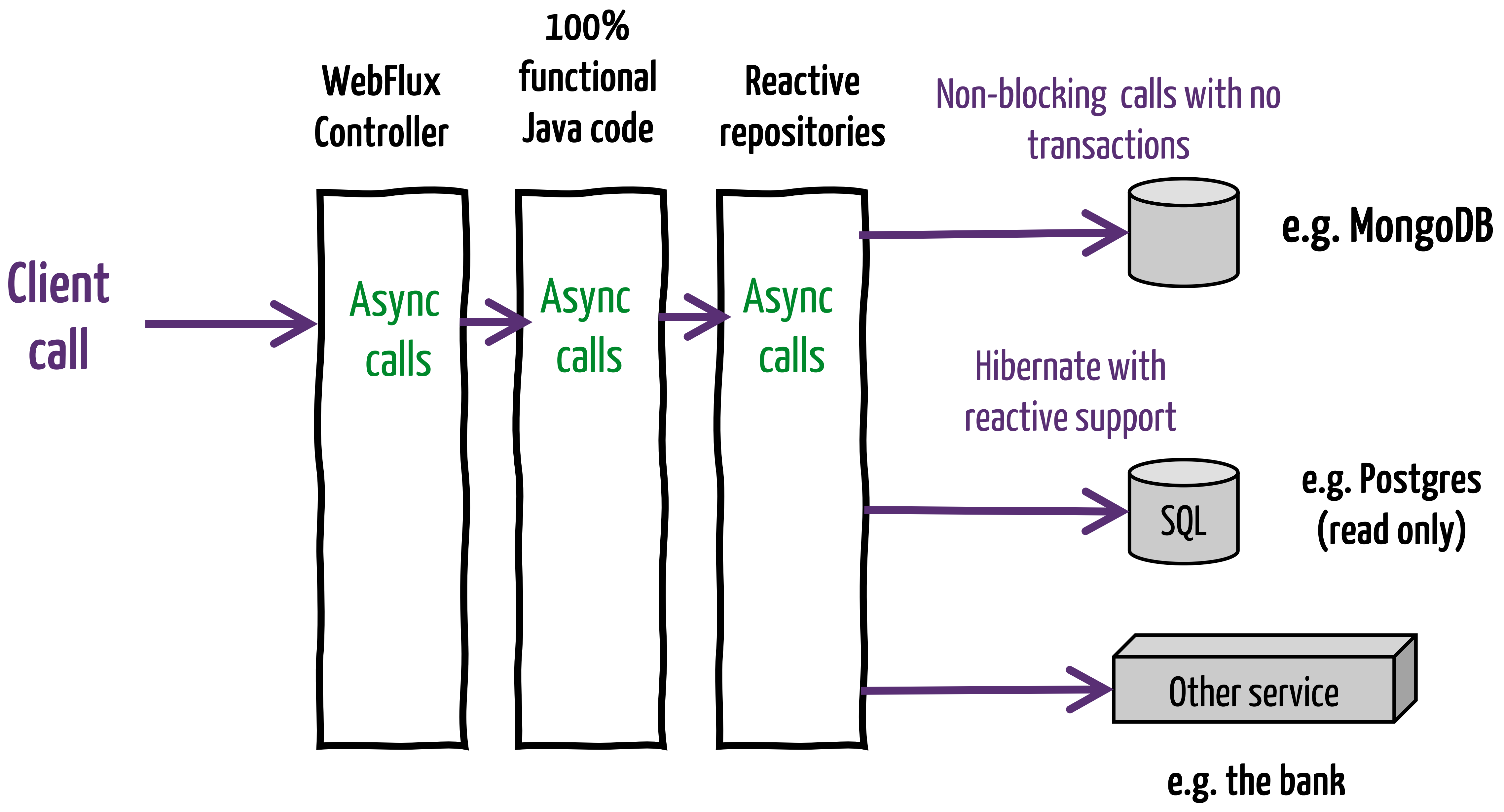
Do Not Use

- A **Non-Blocking** Web stack shipped with Spring 5.0
    - Concurrency with small number of threads and fewer resources
    - Based on Reactive API from the controllers to the repositories
- Useful **ONLY** when the Web stack is **reactive END TO END**
- **Harder to grasp with a really steep learning curve**

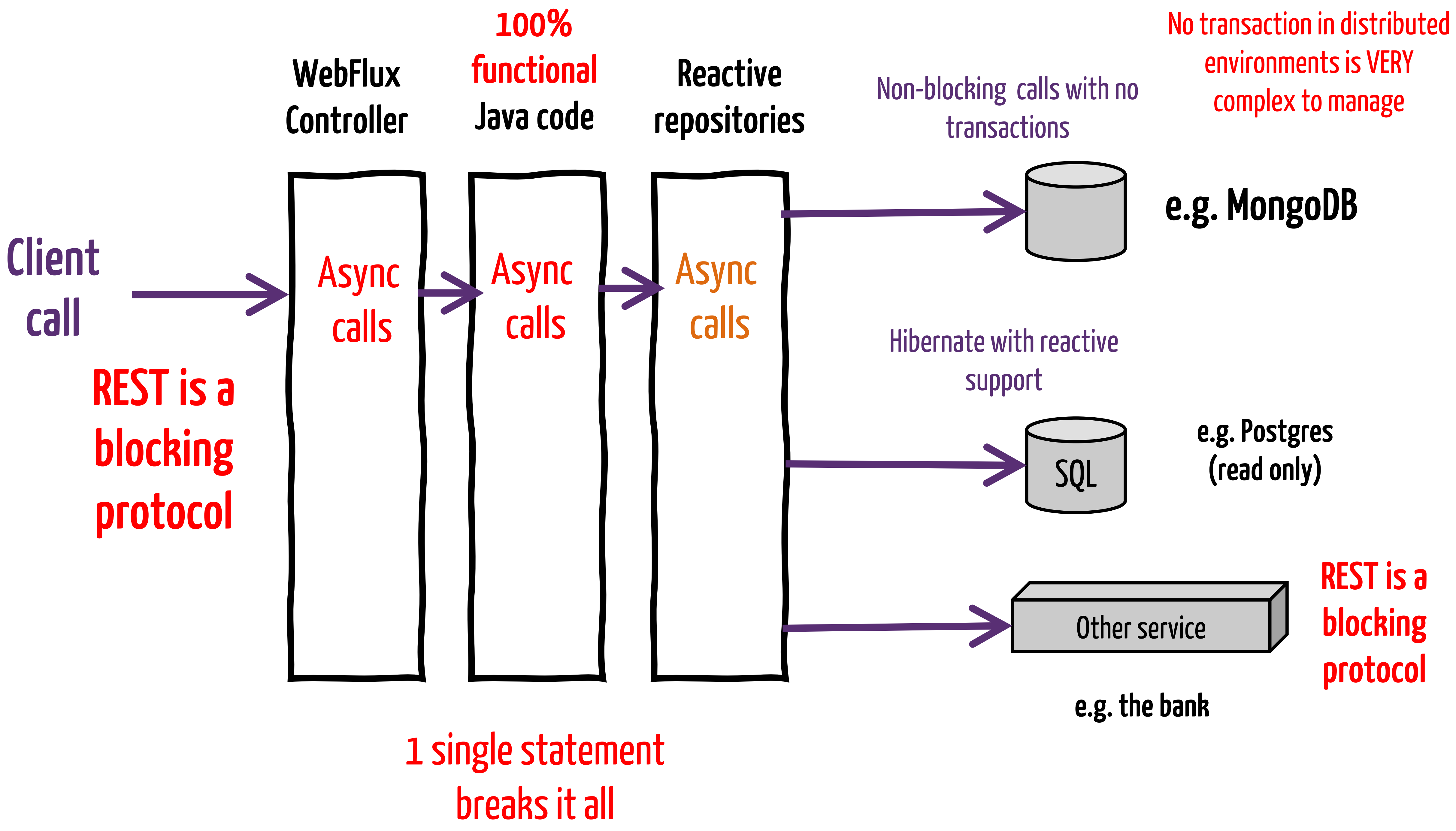
There are already many concepts and technologies in this course

You have to understand the synchronous/blocking behaviour first

Useful **ONLY** when the Web stack is **reactive END TO END**

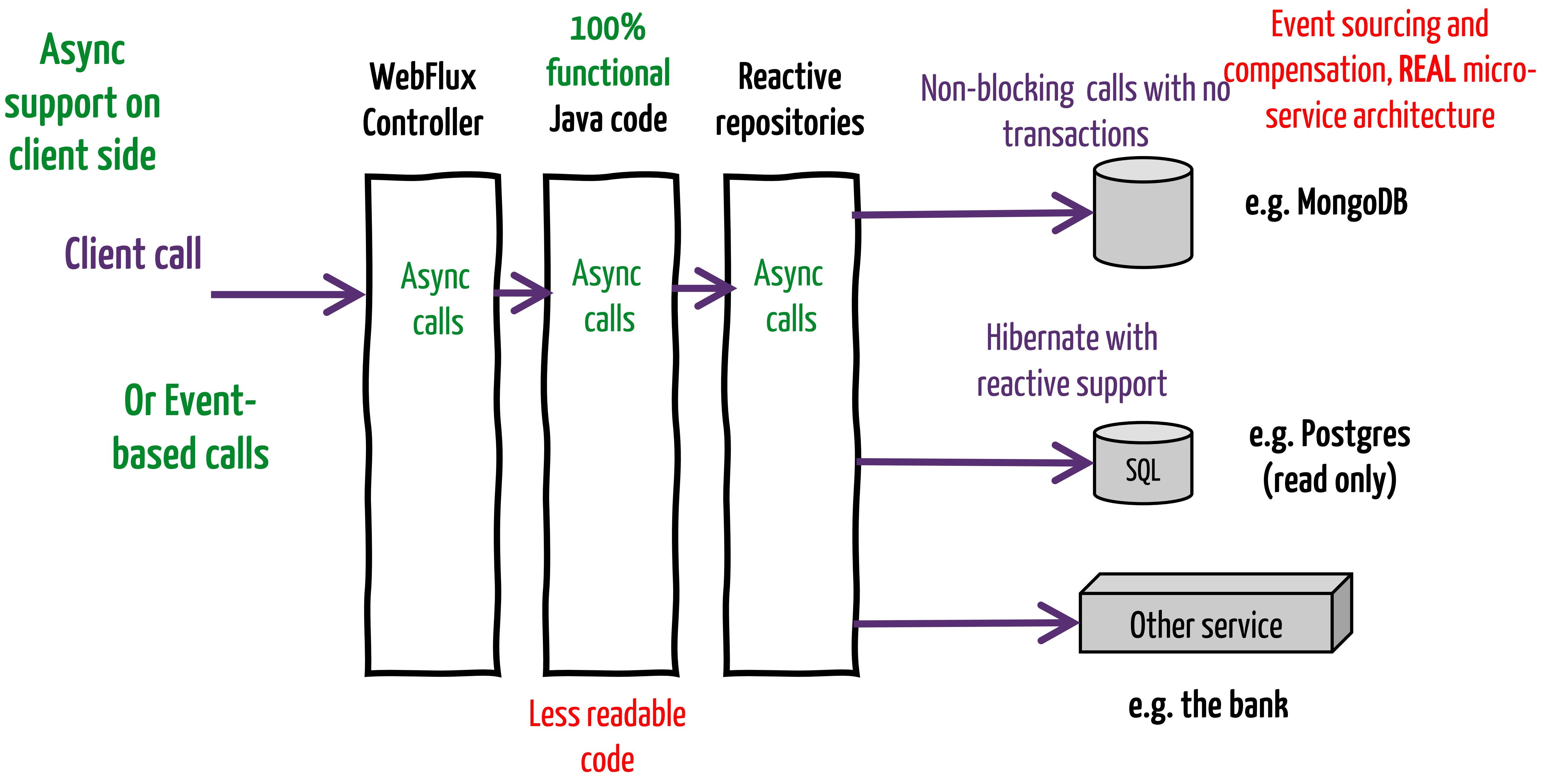


Useful **ONLY** when the Web stack is **reactive** **END TO END**



Client call
REST is a blocking protocol
WebFlux Controller
100% functional Java code
Reactive repositories
Async calls
Async calls
Async calls
Non-blocking calls with no transactions
No transaction in distributed environments is VERY complex to manage
e.g. MongoDB
Hibernate with reactive support
SQL
e.g. Postgres (read only)
Other service
REST is a blocking protocol
e.g. the bank
1 single statement breaks it all

Useful **ONLY** when the Web stack is **reactive END TO END**


Async support on client side
Client call
Or Event-based calls
WebFlux Controller
100% functional Java code
Reactive repositories
Async calls
Async calls
Async calls
Less readable code
Non-blocking calls with no transactions
Event sourcing and compensation, REAL micro-service architecture
e.g. MongoDB
Hibernate with reactive support
SQL
e.g. Postgres (read only)
Other service
e.g. the bank

We are NOT going to use it

RESTful

- REST (REpresentational State Transfer)

- REST-compliant Web services allow the requesting systems to access and manipulate textual representations of web **resources** by using a uniform and predefined set of **stateless** operations

base URL

- media type (JSON is default in Spring)
- standard HTTP methods for interaction

`GET /tickets` - Retrieves a list of tickets

`GET /tickets/12` - Retrieves a specific ticket

`POST /tickets` - Creates a new ticket **(POST can also be used for « control operations »)**

`PUT /tickets/12` - Updates ticket #12

`PATCH /tickets/12` - Partially updates ticket #12

`DELETE /tickets/12` - Deletes ticket #1

REST is not CRUD

What you do 🤬

Think about business interactions

Do CRUD

REST is not CRUD

Golden rules of REST Controllers

1. **Thou shalt not implement business logic in a controller.** The controller is handling interoperability (transfering information back and forth), handling exceptions to return appropriate status codes, and coordinating call to business components. That's all.

2. **Thou shalt not make a controller stateful.** The controller shoud be stateless. It should not keep *conversational state* information between the client and the server (i.e., some information that would oblige to keep the controller object to be the same to handle several distinct calls from the same client to the server). This enables to handle network failure, and to scale horizontally.

A first @RestController

= 1 Spring Component (in the MVC part)

indicates the method is relative to the /recipes path

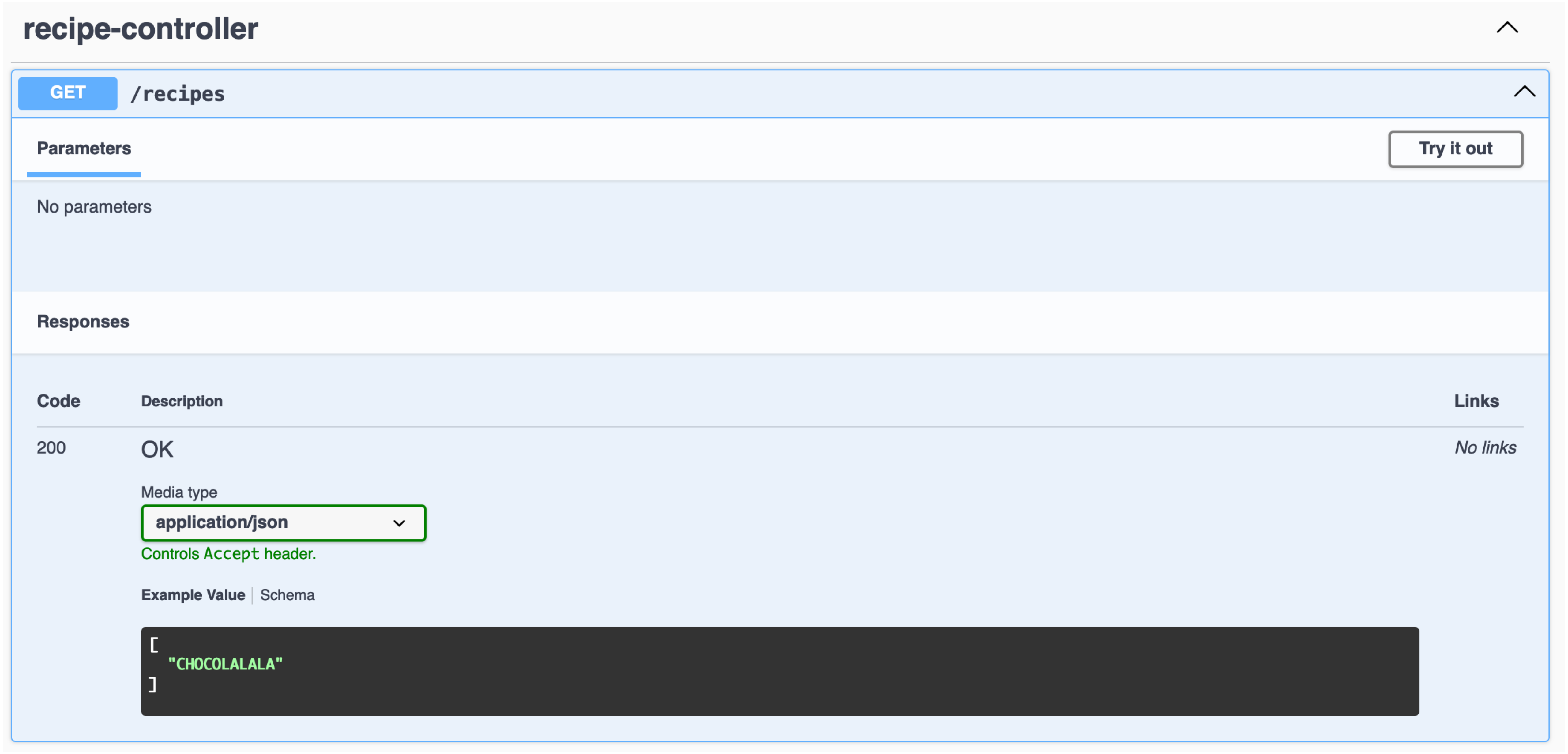
```
@RestController
public class RecipeController {

    public static final String BASE_URI = "/recipes";

    private final CatalogExplorator catalogExp;

    public RecipeController(CatalogExplorator catalogExp) {
        this.catalogExp = catalogExp;
    }

    @GetMapping(path = RecipeController.BASE_URI, produces = APPLICATION_JSON_VALUE)
    public Set<Cookies> listAllRecipes() {
        return catalogExp.listPreMadeRecipes();
    }
}
```

serves the GET verb

Automatic conversion to JSON (array here)

http header Accept -> JSON

/recipes route (from the REST API viewpoint)


recipe-controller

GET /recipes

Parameters

Try it out

No parameters

Responses

Code	Description	Links
200	OK	*No links*

Media type

application/json

Controls Accept header.

Example Value | Schema

```
[
  "CHOCOLALALA"
]
```

Mappings

indicates that all mapping in the class are relative to the /customers path (other mapping will be added after)

```
@RestController
@RequestMapping(path = CustomerCareController.BASE_URI, produces = APPLICATION_JSON_VALUE)
public class CustomerCareController {

    public static final String BASE_URI = "/customers";

    private final CustomerRegistration registry;

    private final CustomerFinder finder;

    @PostMapping(consumes = APPLICATION_JSON_VALUE)
    public ResponseEntity<CustomerDTO> register(@RequestBody @Valid CustomerDTO cusdto) {
```

Implementation of a POST verb

HTTP designates POST as semantically open-ended. It allows the method to take any action, regardless of its repeatability or side effects.

JSON also in the POST body

Body and return codes

Object encapsulating the returned DTO with the http response code

Get the Request Body from JSON to an object

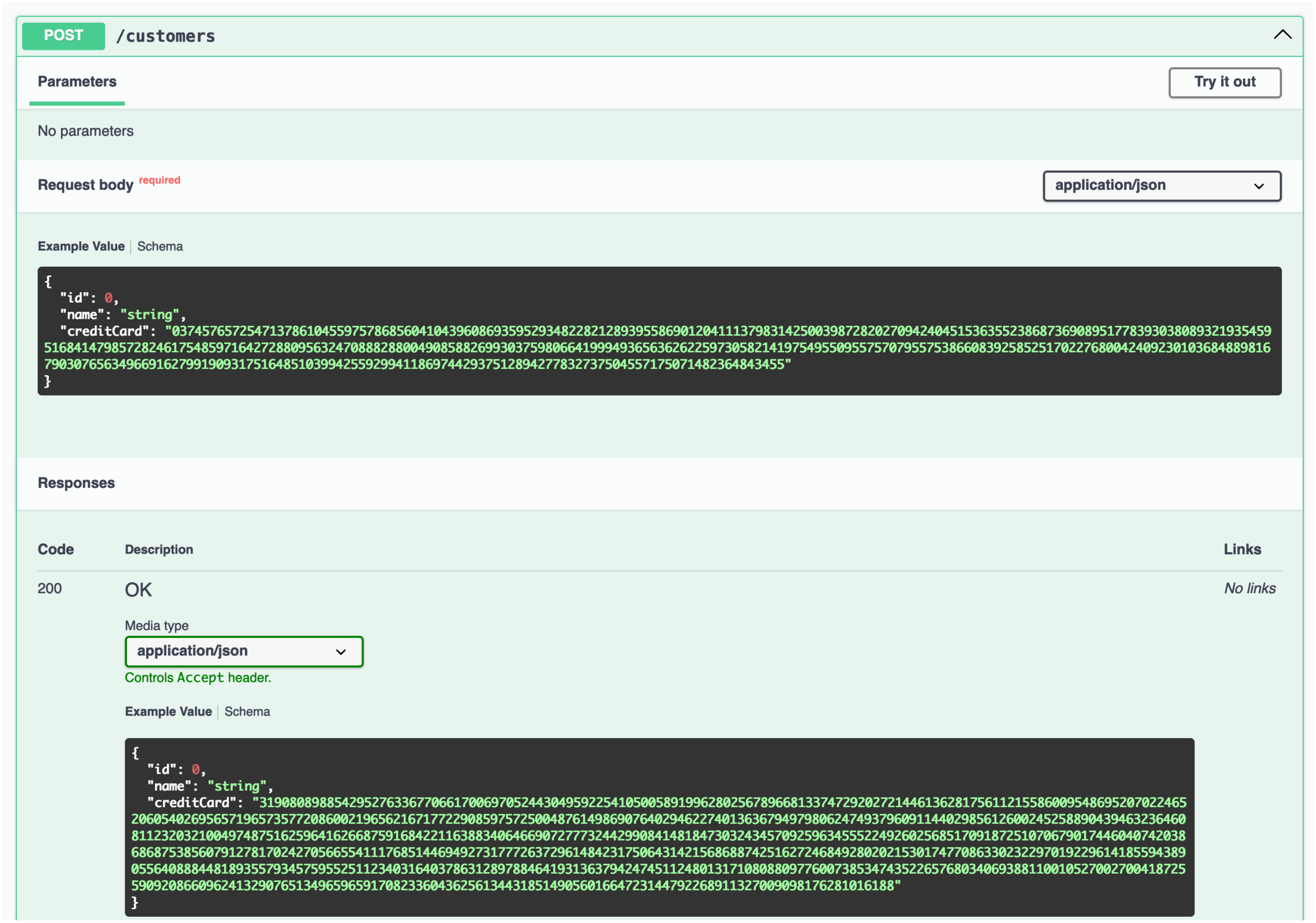
```
@PostMapping(consumes = APPLICATION_JSON_VALUE)
public ResponseEntity<CustomerDTO> register(@RequestBody @Valid CustomerDTO cusdto) {
    // Note that there is no validation at all on the CustomerDto mapped
    try {
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(convertCustomerToDto(registry.register(cusdto.name(), cusdto.creditCard())
    } catch (AlreadyExistingCustomerException e) {
        // Note: Returning 409 (Conflict) can also be seen a security/privacy vulnerability, exp
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}
```

Builder pattern of a return message : the status code + the body

Call to business components

Return a http code for error (here 409)

/customers POST route



```
{
  "id": 0,
  "name": "string",
  "creditCard": "037457657254713786104559757868560410439608693595293482282128939558690120411137983142500398728202709424045153635523868736908951778393038089321935459
51684147985728246175485971642728809563247088828800490858826993037598066419994936563626225973058214197549550955757079557538660839258525170227680042409230103684889816
790307656349669162799190931751648510399425592994118697442937512894277832737504557175071482364843455"
}
```

```
{
  "id": 0,
  "name": "string",
  "creditCard": "3190808988542952763367706617006970524430495922541050058919962802567896681337472920272144613628175611215586009548695207022465
206054026956571965735772086002196562167177229085975725004876149869076402946227401363679497980624749379609114402985612600245258890439463236460
811232032100497487516259641626687591684221163883406466907277732442990841481847303243457092596345552249260256851709187251070679017446040742038
686875385607912781702427056655411176851446949273177726372961484231750643142156868874251627246849280202153017477086330232297019229614185594389
055640888448189355793457595525112340316403786312897884641931363794247451124801317108088097760073853474352265768034069388110010527002700418725
590920866096241329076513496596591708233604362561344318514905601664723144792268911327009098176281016188"
}
```

The DTO (Data Transfer Object) pattern

Objective: avoid tight coupling on data and ensure data integrity/security

- decouple the domain models from the presentation layer (allowing both to change independently)
- encapsulate the serialization logic
- potentially reduce the number of method calls (lowering the network overhead)

Design: simply convert in and out some internal data model to a specific data model for transport outside of the application

Implementation:

- Could be very simple (removing some attributes)
- Could be more complex (building a different representation/model from several objects

JSON Model Mapper can also be seen as DTO implementation, especially with @JSONignore annotations to filter out unwanted attributes

DTO pattern: a SIMPLE application

```
public record CustomerDTO (
    Long id, // expected to be empty when POSTing the creation of Customer, and containing the Id when retur
    @NotBlank(message = "name should not be blank") String name,
    @Pattern(regexp = "\\d{10}+", message = "credit card should be exactly 10 digits") String creditCard) {
}
```

Between cli and backend

Inside the backend

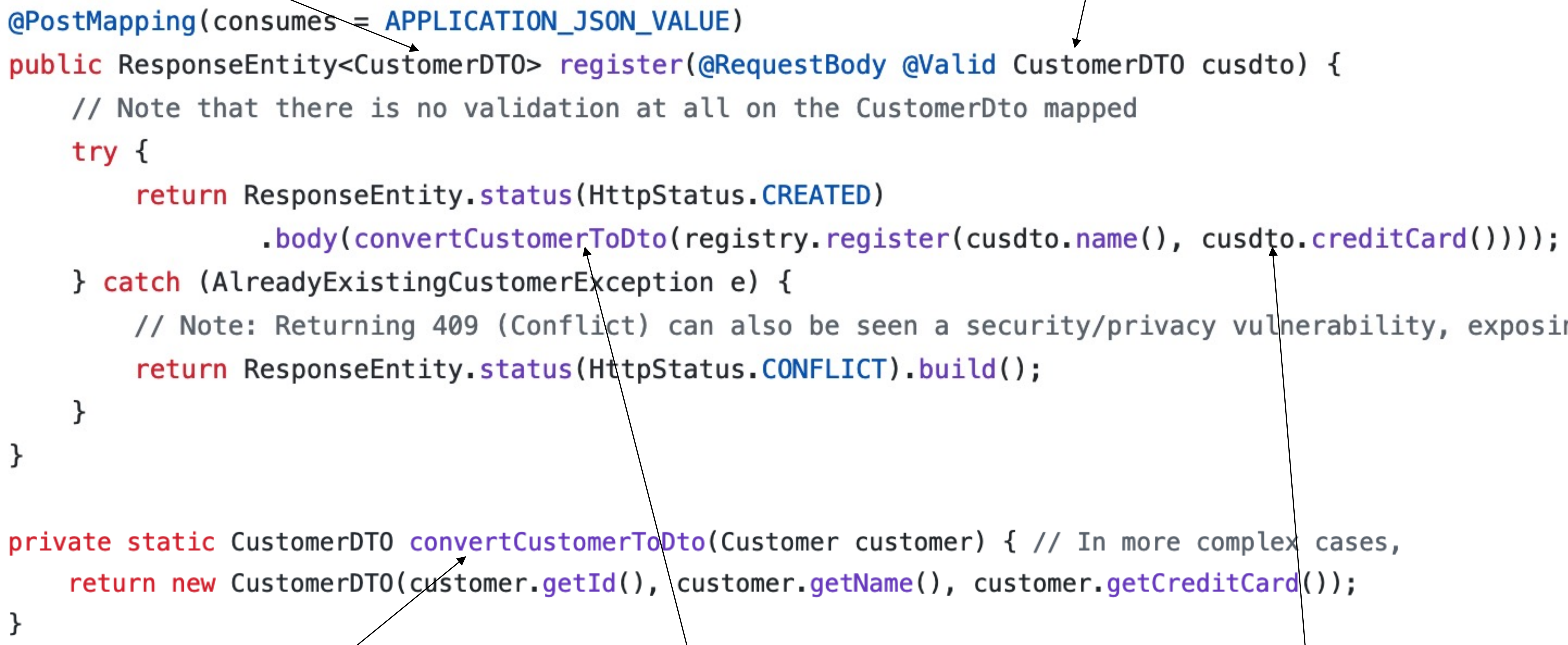
```
public class CliCustomer {

    private Long id;
    private String name;
    private String creditCard;
```

```
public class Customer {

    @Id
    @GeneratedValue
    private Long id; // Whether Long/Int or UUID are better primary keys,

    @NotBlank
    @Column(unique = true)
    private String name;

    @Pattern(regexp = "\\d{10}+", message = "Invalid creditCardNumber")
    private String creditCard;

    @OneToMany(cascade = {CascadeType.REMOVE}, fetch = FetchType.LAZY, ma
    private Set<Order> orders = new HashSet<>();

    @ElementCollection(fetch = FetchType.EAGER)
    private Set<Item> cart = new HashSet<>();
```

DTO pattern: a SIMPLE application

CustomerDTO returned (converted in JSON)

CustomerDTO with no ID passed in POST



```
@PostMapping(consumes = APPLICATION_JSON_VALUE)
public ResponseEntity<CustomerDTO> register(@RequestBody @Valid CustomerDTO cusdto) {
    // Note that there is no validation at all on the CustomerDto mapped
    try {
        return ResponseEntity.status(HttpStatus.CREATED)
                .body(convertCustomerToDto(registry.register(cusdto.name(), cusdto.creditCard())));
    } catch (AlreadyExistingCustomerException e) {
        // Note: Returning 409 (Conflict) can also be seen a security/privacy vulnerability, exposin
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}

private static CustomerDTO convertCustomerToDto(Customer customer) { // In more complex cases,
    return new CustomerDTO(customer.getId(), customer.getName(), customer.getCreditCard());
}
```

Conversion method

Call to conversion method to return the DTO

Extraction from the DTO to use data in business logic

Path variables (CartController)

Path variable

Using it as a method parameter

```
public static final String CART_URI = "/{customerId}/cart";
```

```
@PostMapping(path = CART_URI, consumes = APPLICATION_JSON_VALUE)
public ResponseEntity<Item> updateCustomerCart(@PathVariable("customerId") Long customerId, @RequestBody Item it)
    return ResponseEntity.ok(cart.update(customerId, it)); // Item is used as a DTO in and out here...
}
```

```
throws CustomerIdNotFoundException, NegativeQuantityException {
```

Wait!? What about exceptions?

```
@PostMapping(path = CART_URI, consumes = APPLICATION_JSON_VALUE)
public ResponseEntity<Item> updateCustomerCart(@PathVariable("customerId") Long customerId, @RequestBody Item it)
                throws CustomerIdNotFoundException, NegativeQuantityException {
```

Default behaviour: uncaught exception

-> 500 status code returned

Exception handlers

```
public record ErrorDTO (String error, String details) {
}
```

```
@RestControllerAdvice(assignableTypes = {CustomerCareController.class, CartController.class})
public class GlobalControllerAdvice {

@ExceptionHandler({CustomerIdNotFoundException.class})
@ResponseStatus(HttpStatus.NOT_FOUND)
public ErrorDTO handleExceptions(CustomerIdNotFoundException e) {
    return new ErrorDTO("Customer not found", e.getId() + " is not a valid customer Id");
}

@ExceptionHandler({NegativeQuantityException.class})
@ResponseStatus(HttpStatus.FORBIDDEN)
public ErrorDTO handleExceptions(NegativeQuantityException e) {
    return new ErrorDTO("Attempting to update the cookie quantity to a negative value",
            "from Customer " + e.getName() + " with cookie " + e.getCookie() +
                    " leading to quantity " + e.getPotentialQuantity());
}
```

The exception handling code is separated from the business logic, which itself is not polluted

The exception handling code can be reused, and it is reused: The CustomerIdNotFoundException may be thrown in all route implementation of the corresponding path.

Parameters and path variables

URL: user/1234/invoices?date=12-05-2013

Gets the invoices of the user with id 1234 for the date of December 5th, 2013

```
@RequestMapping(value="/user/{userId}/invoices", method = RequestMethod.GET)
public List<Invoice> listUsersInvoices(
            @PathVariable("userId") int user,
            @RequestParam(value = "date", required = false) Date dateOrNull) {
  ...
}
```

PathVariable: to obtain some placeholder in an URI Template

Here the variable is named as it is different from the method parameter

Can be used in any RequestMethod

RequestParam: to obtain a parameter from the URL

Can be optional

https://github.com/CookieFactoryInSpring/simpleTCFS