TCFS: The Cookie Factory (in Spring)

Philippe Collet

pictures: sxc.hu

https://github.com/CookieFactoryInSpring/simpleTCFS

The Cookie Factory... in Spring (aka the simple TCFS)

- Author: Philippe Collet
- Author: Nassim Bounouas
- Reviewer: Mireille Blay-Fornarino
- Reviewer: Anne-Marie Déry
- Reviewer: Nikita Rousseau
- some code and doc borrowed from the original Cookie Factory by Sebastien Mosser, last fork being https://github.com/collet/4A_ISA_TheCookieFactory

This case study is used to illustrate the different technologies involved in the *Introduction to Software Architecture* course given at Polytech Nice - Sophia Antipolis at the graduate level. This demonstration code requires the following software to run properly:

- Build & Spring environment configuration: Maven >=3.9 (provided maven wrapper set to 3.9.11)
- Spring/Java implementation language: Java=21 or above (Java language level is set to Java 21), SpringBoot 3.5.9
- Spring-Shell 3.4.1 for the cli (same SpringBoot/Java underlying versions)
- NestJS 11 (node 24+, npm 11+)
- Docker Engine (with compose) >= 29.x

Product vision

The Cookie Factory (TCF) is a major bakery brand in the USA. The *Cookie on Demand* (CoD) system is an innovative service offered by TCF to its customer. They can order cookies online thanks to an application, and select when they'll pick-up their order in a given shop. The CoD system ensures to TCF's happy customers that they'll always retrieve their pre-paid warm cookies on time.

Chapters

1. Architecture and deployment
2. Business components
3. Controllers
4. Testing
5. Persistence
6. AOP, logging, and monitoring

What are the “elements” in TCF?

How are they related to each others?

WebApp

Mobile

Interoperability

Customer Care

Kitchen

Order Tracking

Cashier

TCF

Bank (external)

https://github.com/CookieFactoryInSpring/simpleTCFS

/ chapters / Architecture.md

Answers

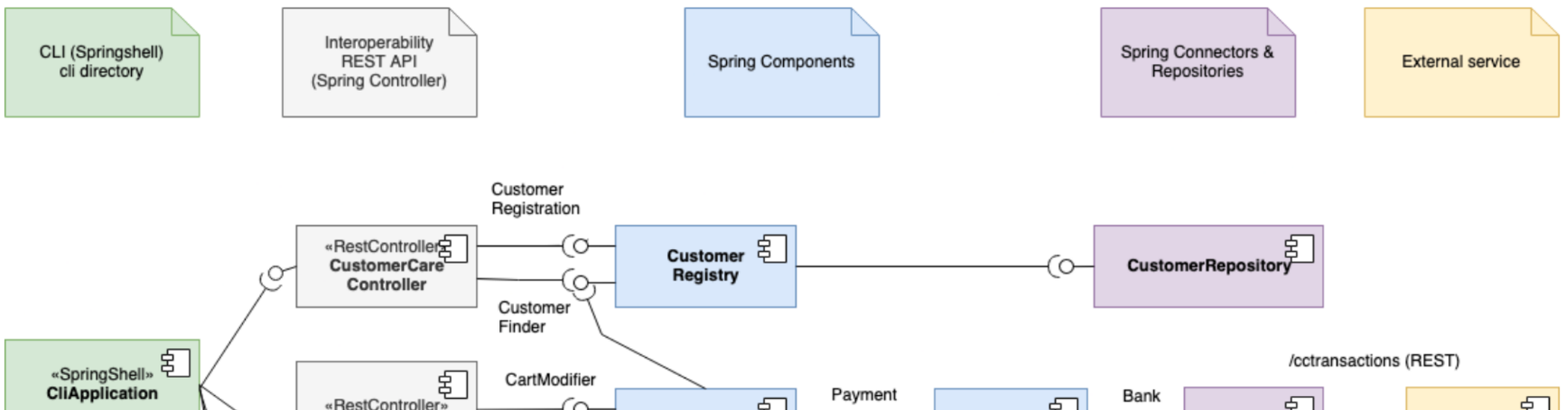
Architecture and Deployment

- Author: Philippe Collet

Components assembly

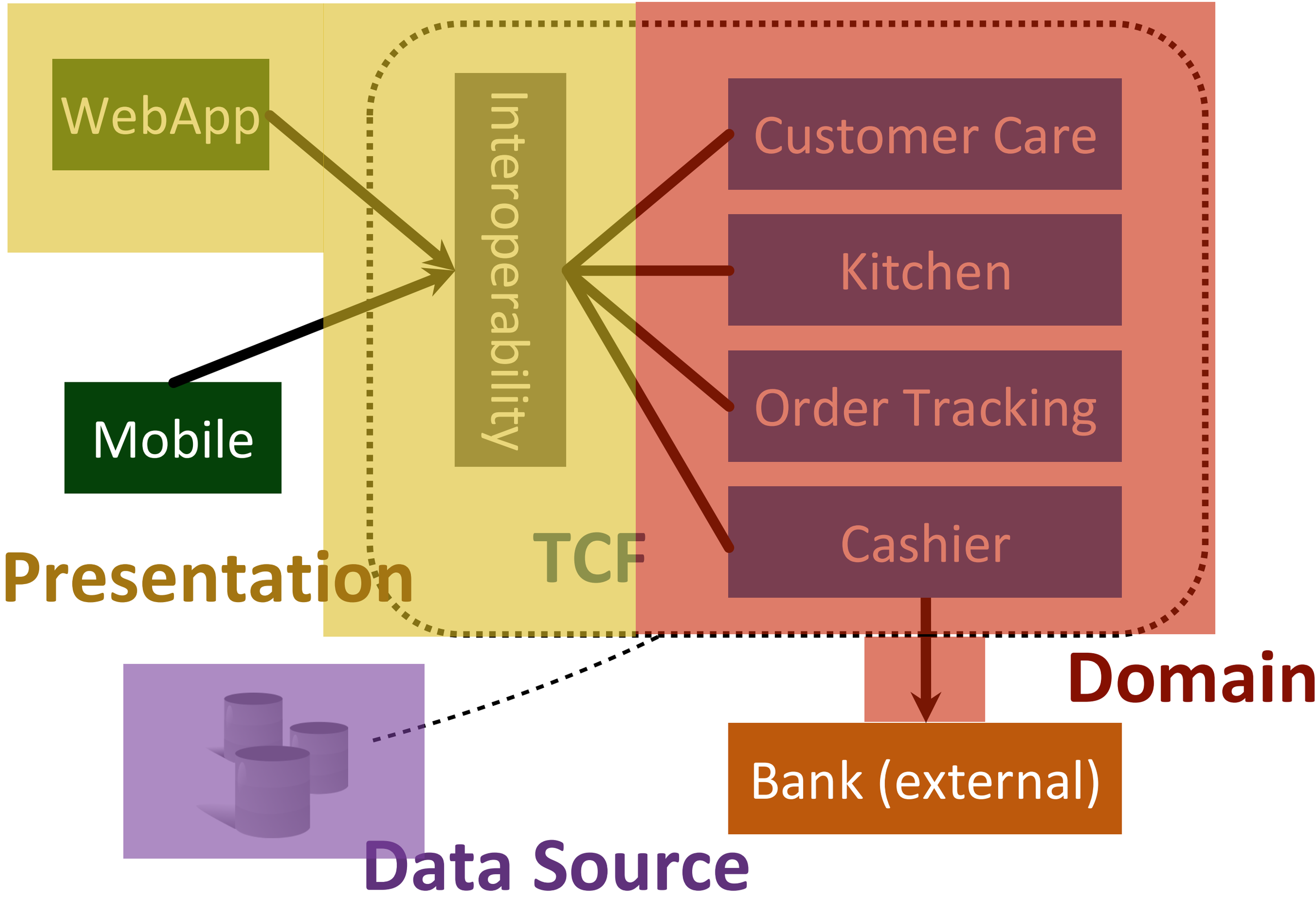
The system is defined as layers:

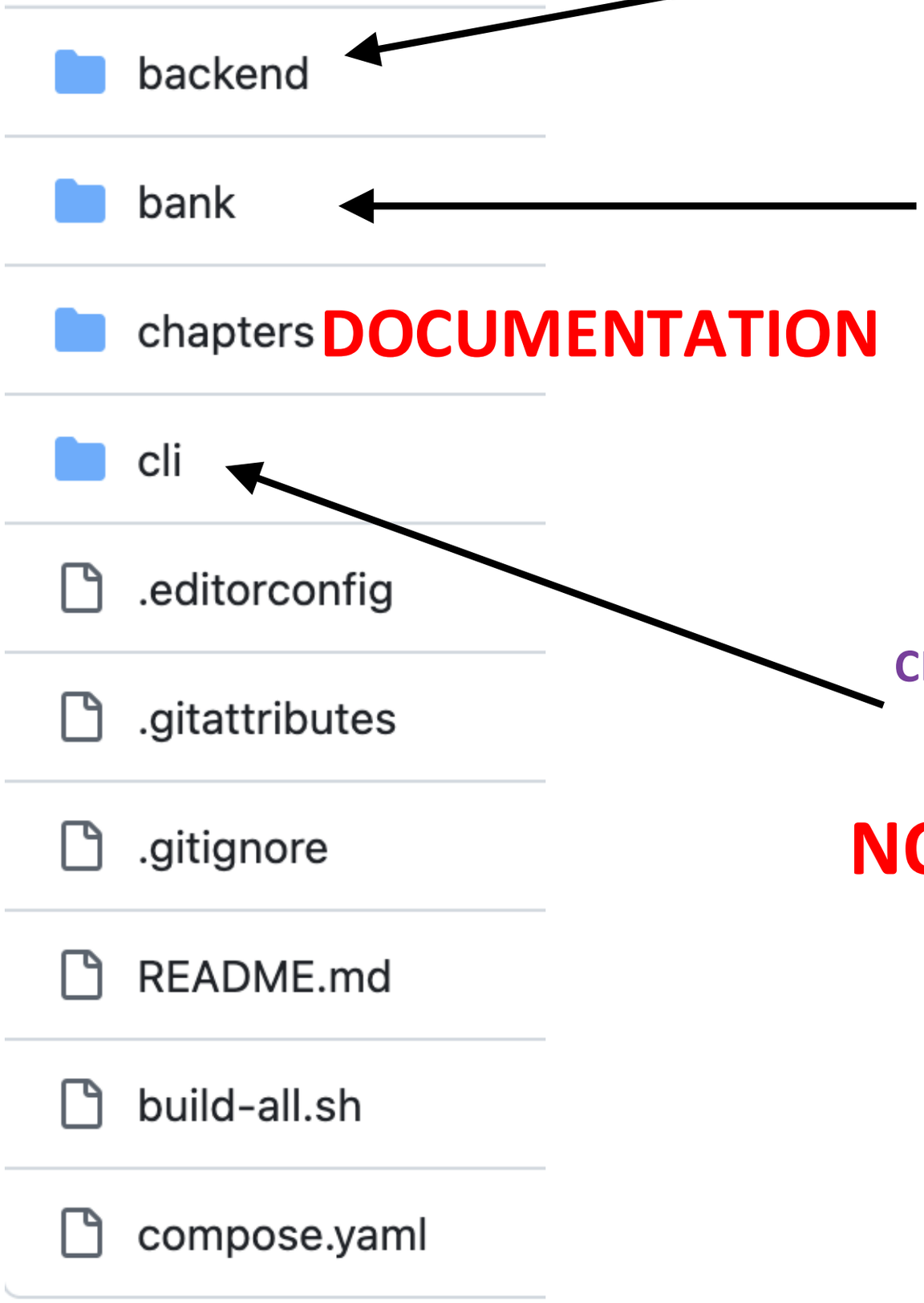
- A remote client (green), that will run on each customer's device (simulated here by a command line interface);
- A Spring kernel (blue), implementing the business logic of the CoD system;
- An interoperability layer (grey) between the client and the kernel, implemented as REST Controllers in Spring (in the same backend project as the blue components);
- A connector and repositories layer (purple) between the components and the data provider, connectors are for external partners, repositories are for uniform data access;
- An external partner (orange), communicating with the CoD system through a REST API as well.



Contents of the ≠ layers in TCF?

How to chose between layers?


WebApp
Mobile
Interoperability
Customer Care
Kitchen
Order Tracking
Cashier
Bank (external)
Presentation
TCF
Domain
Data Source



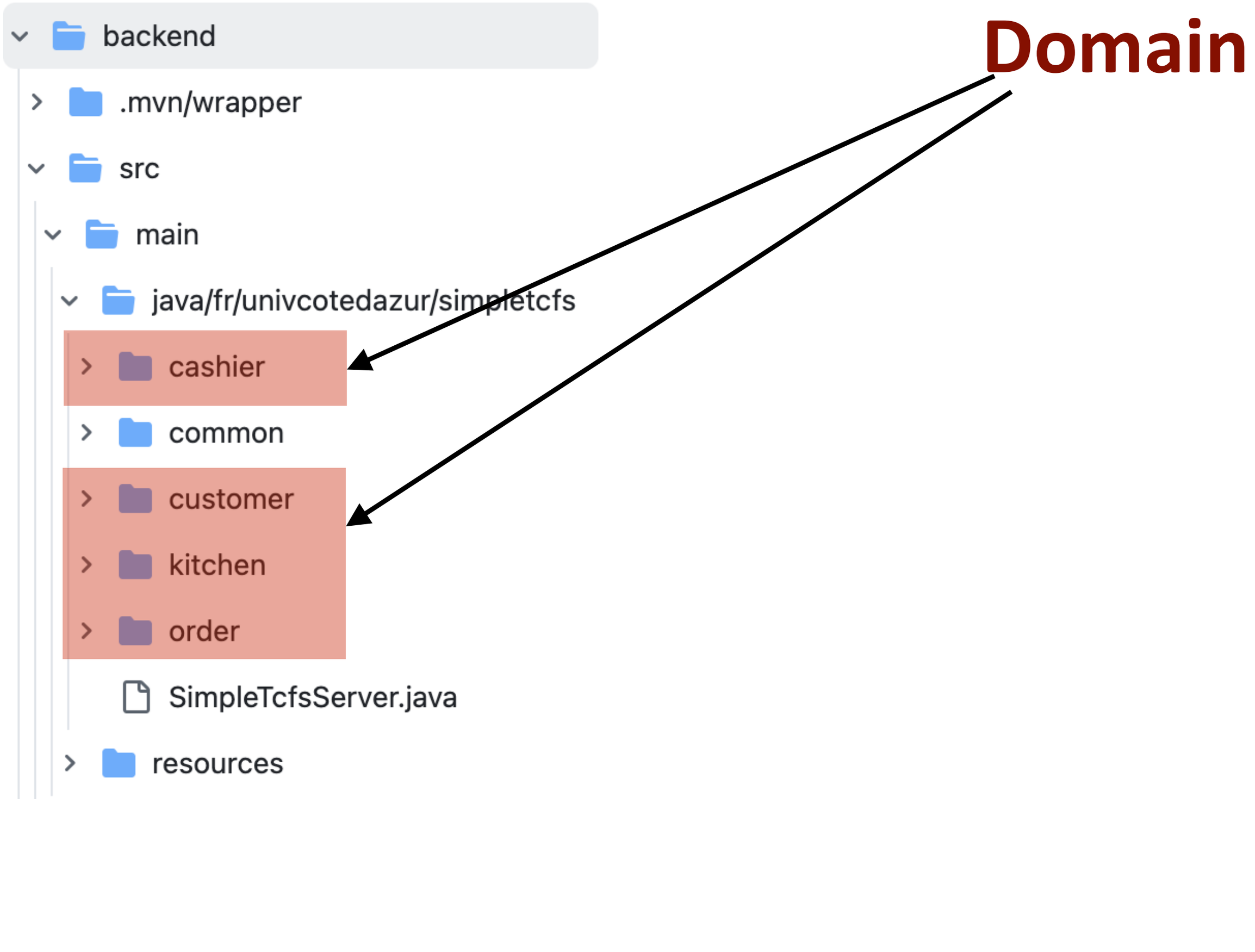
Spring backend (Spring MVC for REST + Spring Components as you know them)

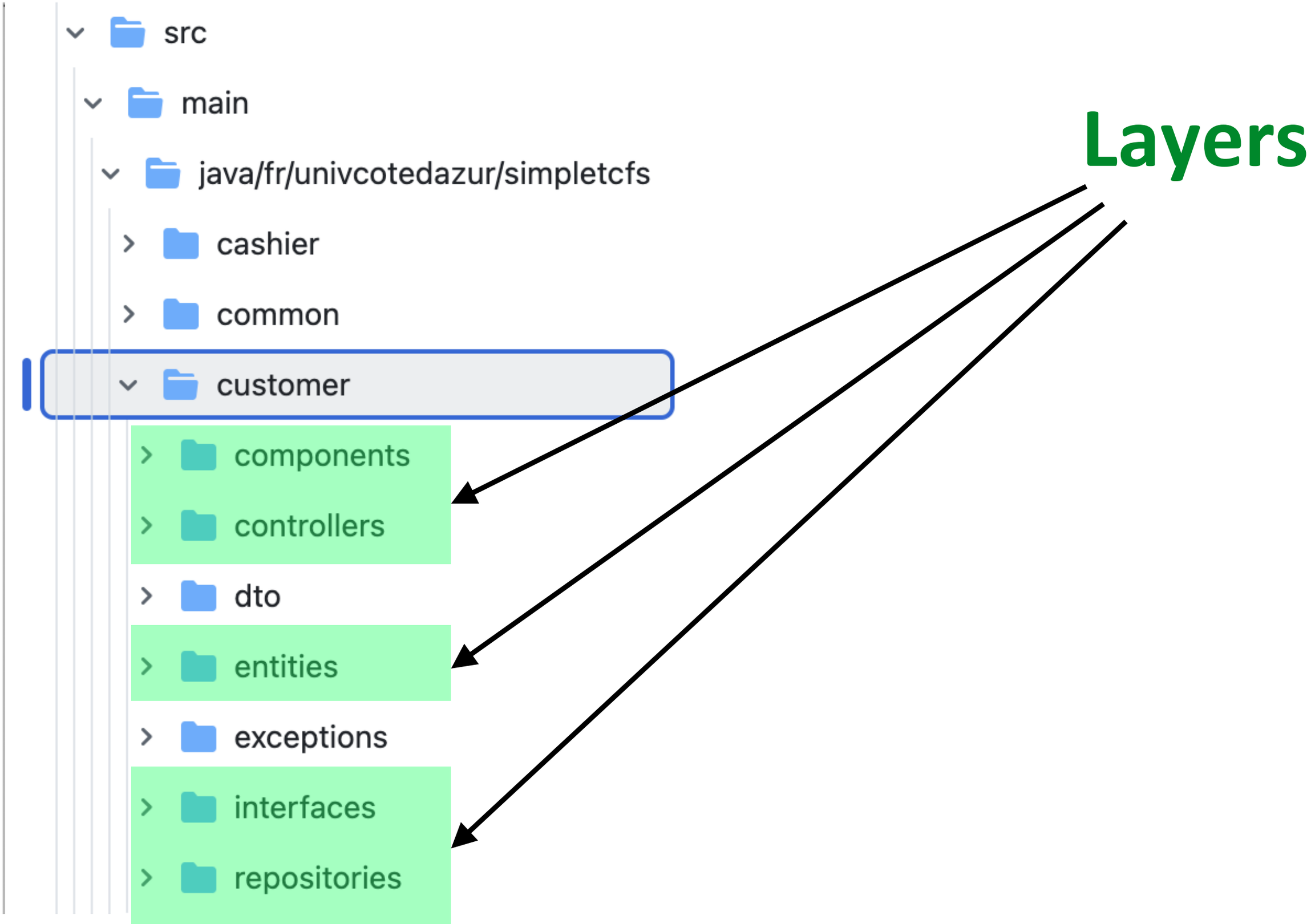
Ultra simple bank in NestJS as an external system

DOCUMENTATION

CLI (Command Line Interface) implemented in Spring Shell

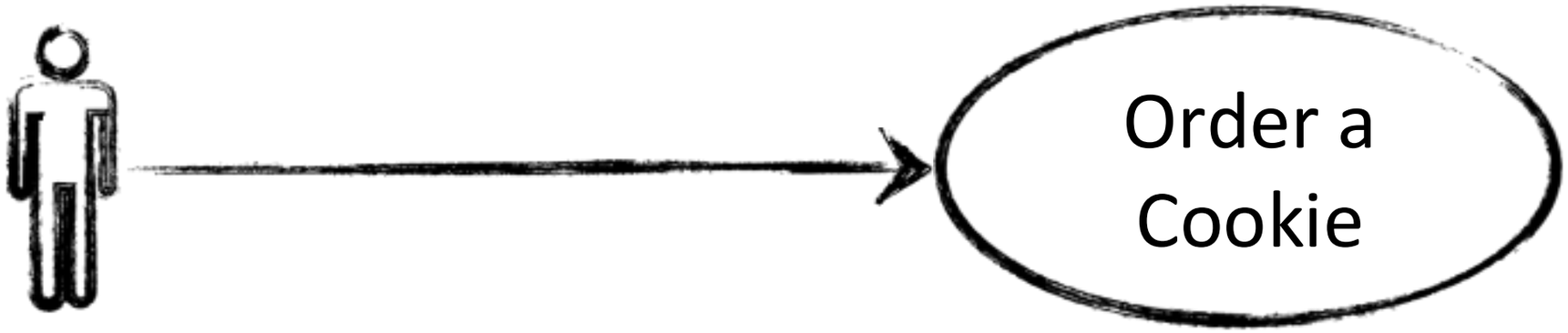
NO GUI/FRONT END


Domain
backend
.mvn/wrapper
src
main
java/fr/univcotedazur/simpletcfs
cashier
common
customer
kitchen
order
SimpleTcfsServer.java
resources


src
main
java/fr/univcotedazur/simpletcfs
cashier
common
customer
components
controllers
dto
entities
exceptions
interfaces
repositories
Layers

TCF MVP

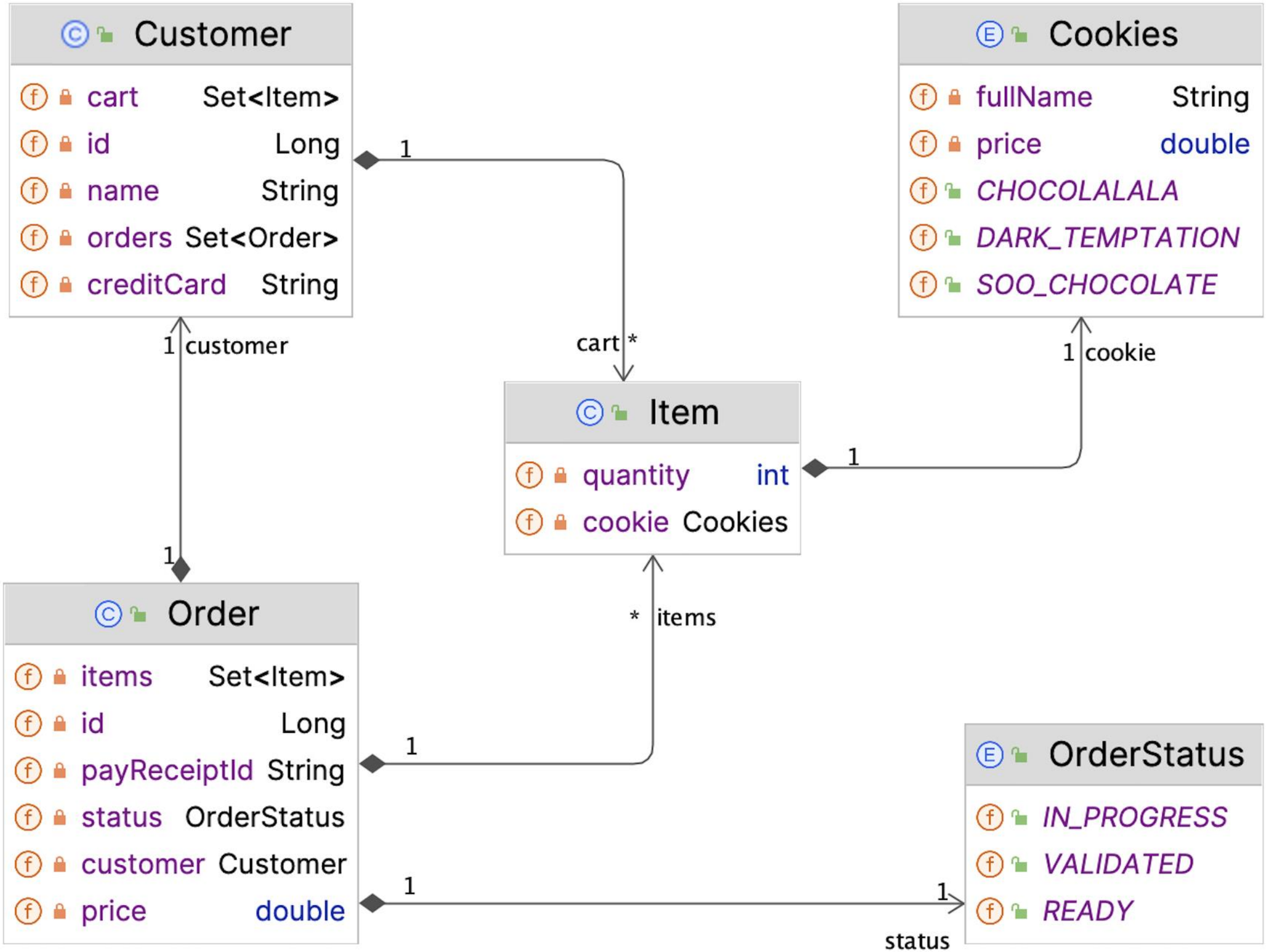
Minimal & Viable Use Case ?



customer

TCF MVP

Business Objects (≠ “objects” as in pure Java)


Customer
cart Set<Item>
id Long
name String
orders Set<Order>
creditCard String
Cookies
fullName String
price double
CHOCOLALALA
DARK_TEMPTATION
SOO_CHOCOLATE
Item
quantity int
cookie Cookies
Order
items Set<Item>
id Long
payReceiptId String
status OrderStatus
customer Customer
price double
OrderStatus
IN_PROGRESS
VALIDATED
READY
1
cart *
1 customer
1 cookie
1
1
* items
1
1
1
status

```java
@Entity
public class Customer {

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

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

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

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

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

Annotations related to persistence (not important right now)

TCF MVP

Functional Interfaces for TCF? Components?

Functional interfaces

To deliver the expected features, the coD system defines the following interfaces:

- `CartModifier` : operation to modify a given customer's cart, by updating the number of cookies in it, and to retrieve the contents of the cart
- `CartProcessor` : operations for computing the cart's price and validating it to process the associated order;
- `CustomerFinder` : a *finder* interface to retrieve a customer based on her identifier (here simplified to her name);
- `CustomerRegistration` : operations to handle customer's registration (users profile, ...)
- `CatalogueExploration` : operations to retrieve recipes available for purchase in the CoD;
- `OrderCreator` : operations to create an order from the customer;
- `OrderFinder` : a *finder* interface to retrieve an order based on its identifier, to follow its status;
- `OrderModifier` : operations to modify the order's status
- `OrderCooking` : operations to process an order (kitchen order lifecycle management);
- `Payment` : operations related to the payment of a given cart's contents;
- `Bank` : operations that act as proxies to the external bank service.

Provided Interfaces

The component is very basic, still we apply interface segregation with two interfaces:

- `CartModifier`: operations to modify a given customer's cart, like adding or removing cookies, and to retrieve the contents of the cart;

Ids, DTOs or objects from the business model

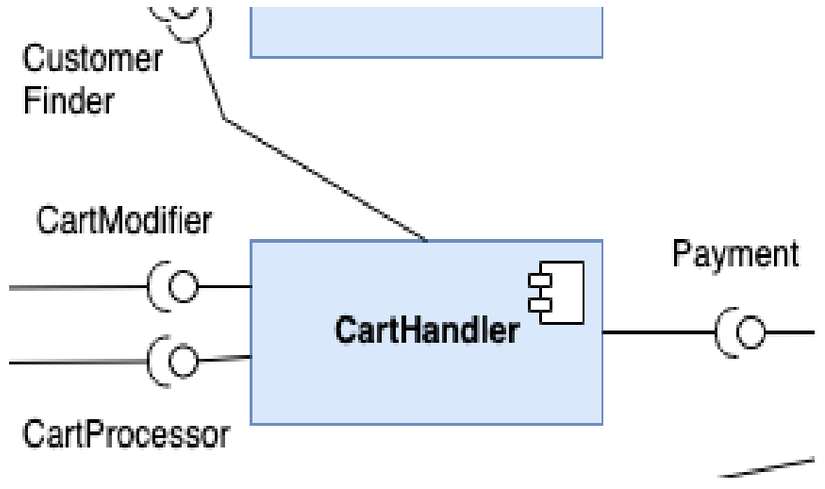
```
public interface CartModifier {

    Item update(Long customerId, Item it) throws NegativeQuantityException, CustomerIdNotFoundExce

    Set<Item> cartContent(Long customerId) throws CustomerIdNotFoundException;
}
```

- `CartProcessor`: operations for computing the cart's price and validating it to process the associated order;

```
public interface CartProcessor {

    double cartPrice(Long customerId) throws CustomerIdNotFoundException;

    Order validate(Long customerId) throws PaymentException, EmptyCartException, CustomerIdNotFoun
}
```

```java
public interface CartModifier {

    Item update(Long customerId, Item it) throws NegativeQuantityException, CustomerIdNotFoundException;

    Set<Item> cartContent(Long customerId) throws CustomerIdNotFoundException;

}
public interface CartProcessor {

    double cartPrice(Long customerId) throws CustomerIdNotFoundException;

    Order validate(Long customerId) throws PaymentException, EmptyCartException, CustomerIdNotFoundException;

}
```

```java
@Service
public class CartHandler implements CartModifier, CartProcessor {

    private static final Logger LOG = LoggerFactory.getLogger(CartHandler.class);

    private final Payment payment;

    private final CustomerFinder customerFinder;

    @Autowired
    public CartHandler(Payment payment, CustomerFinder customerFinder) {
        this.payment = payment;
        this.customerFinder = customerFinder;
    }
```



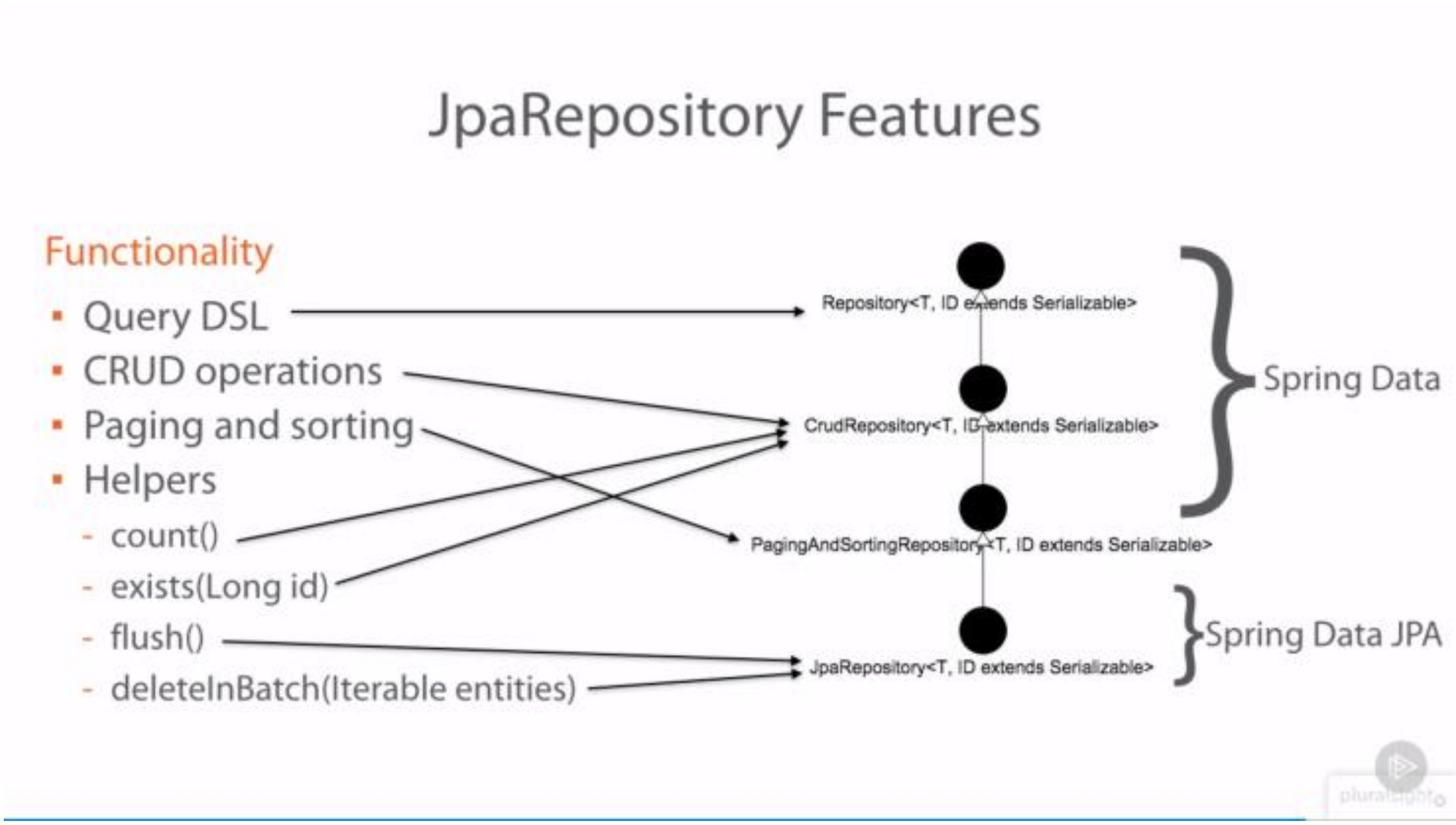

```
public interface CustomerRegistration {

    Customer register(String name, String creditCard)
            throws AlreadyExistingCustomerException;
}
```

```
public interface CustomerFinder {

    Optional<Customer> findByName(String name);

    Optional<Customer> findById(Long id);

    Customer retrieveCustomer(Long customerId) throws CustomerIdNotFoundException;

    List<Customer> findAll();

}
```

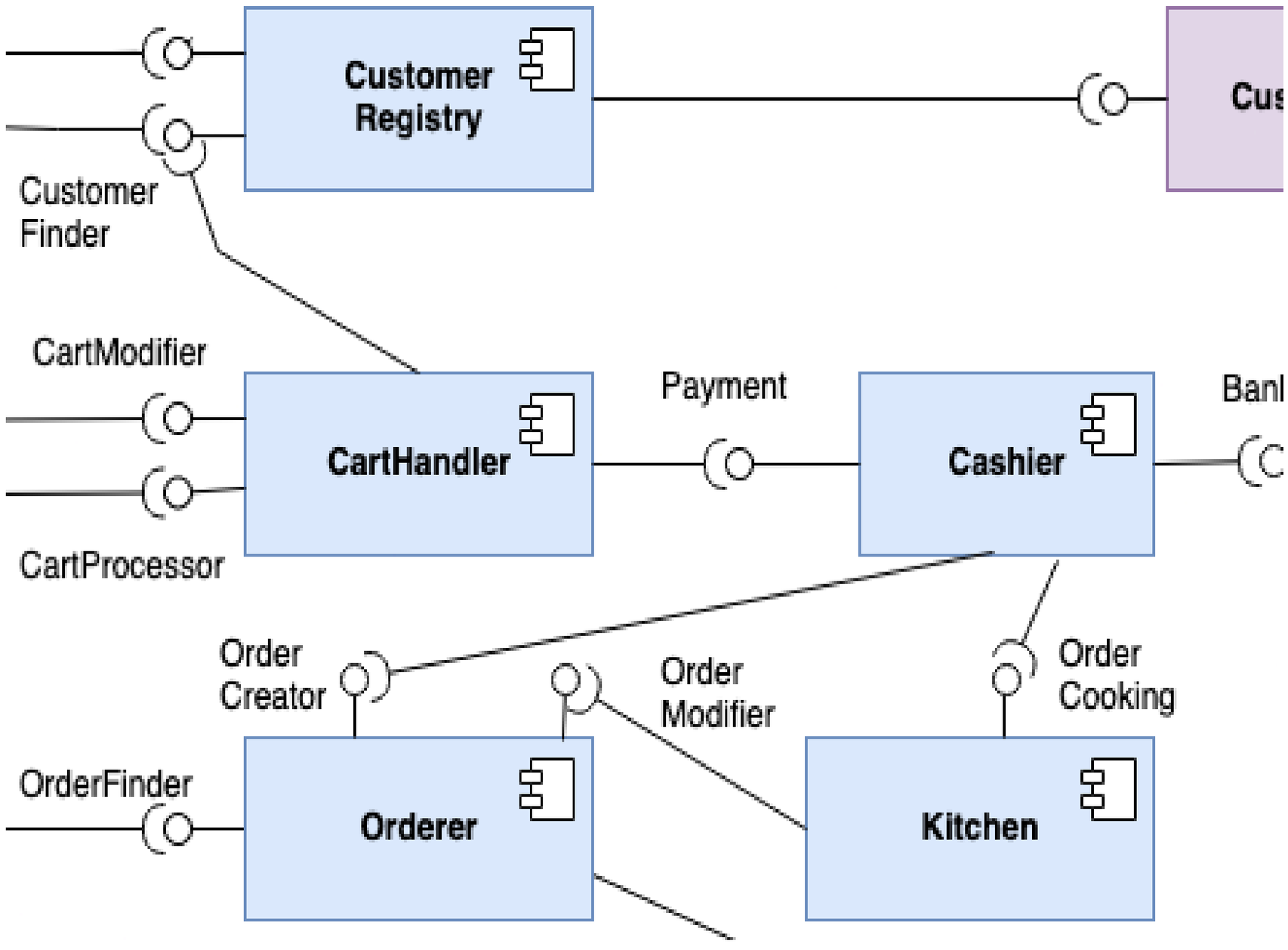
```
@Service
public class CustomerRegistry implements CustomerRegistration, CustomerFinder {

    private final CustomerRepository customerRepository;

    @Autowired // annotation is optional since Spring 4.3 if component has only one constructor
    public CustomerRegistry(CustomerRepository customerRepository) {
        this.customerRepository = customerRepository;
    }
```

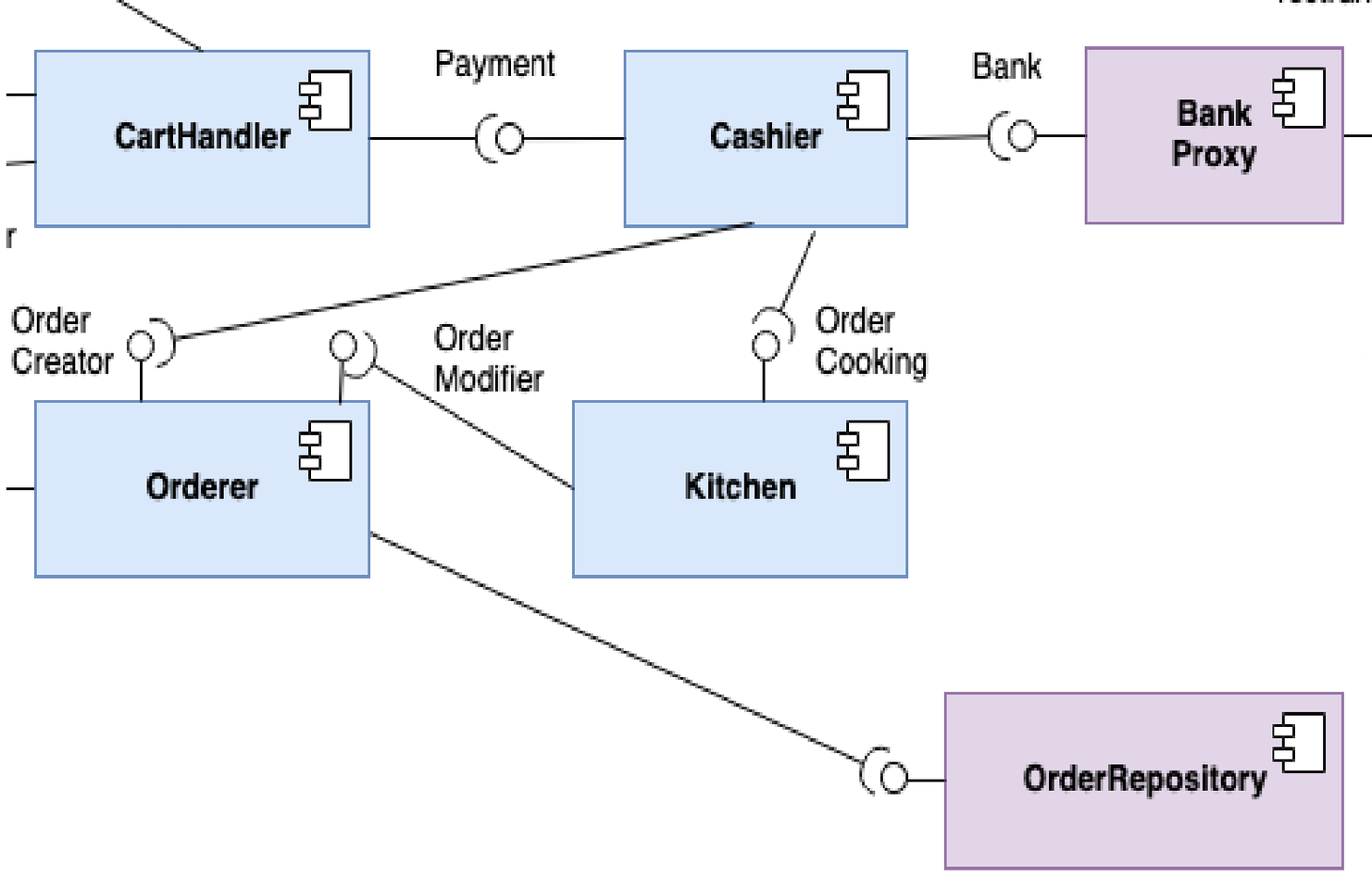
```
@Override
@Transactional
public Customer register(String name, String creditCard)
        throws AlreadyExistingCustomerException {
    if (findByName(name).isPresent())
        throw new AlreadyExistingCustomerException(name);
    Customer newcustomer = new Customer(name, creditCard);
    return customerRepository.save(newcustomer);
}

@Override
@Transactional(readOnly = true)
public Optional<Customer> findByName(String name) {
    return customerRepository.findCustomerByName(name);
}

@Override
@Transactional(readOnly = true)
public Optional<Customer> findById(Long id) {
    return customerRepository.findById(id);
}
```


```
@Repository
public interface CustomerRepository extends JpaRepository<Customer, Long> {

    Optional<Customer> findCustomerByName(String name);

}
```



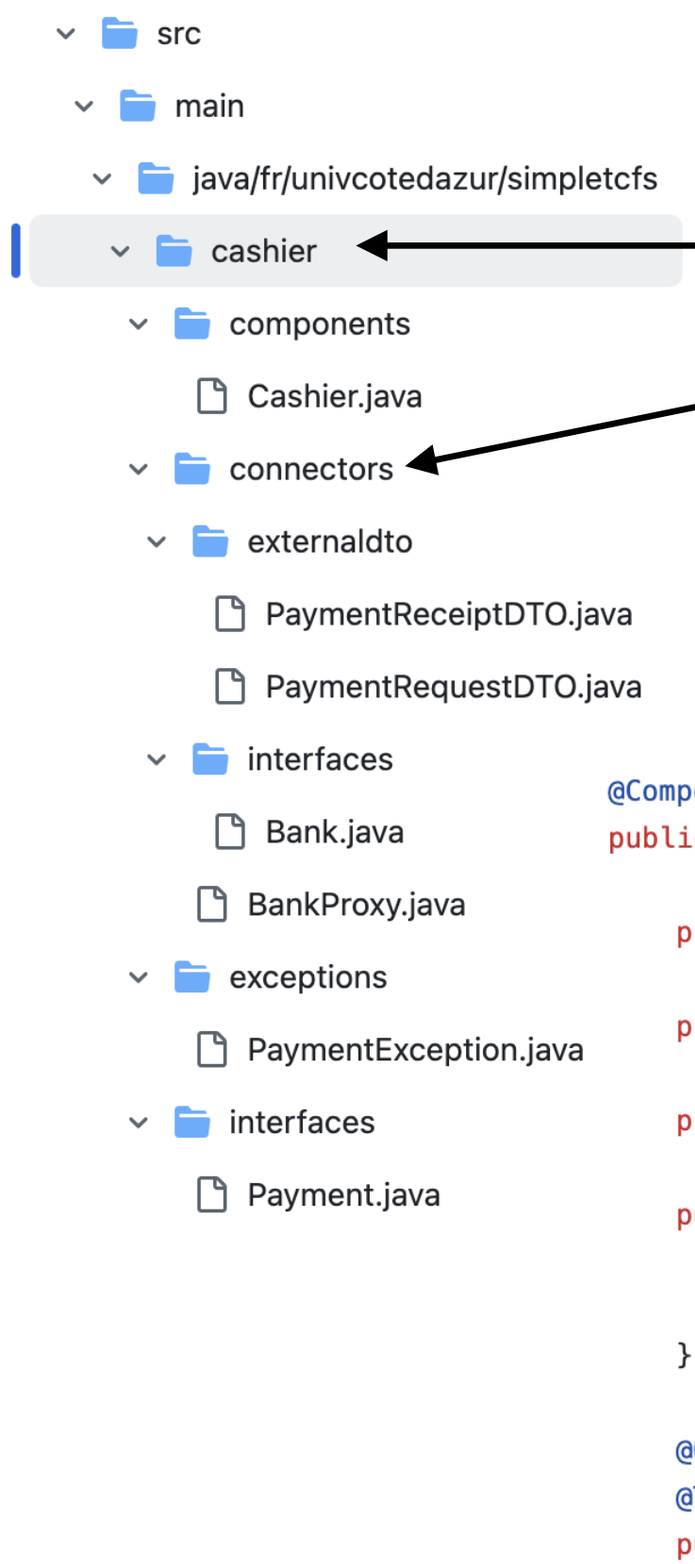
```java
@Service
public class CartHandler implements CartModifier, CartProcessor {


@Override
@Transactional
public Order validate(Long customerId) throws PaymentException, EmptyCartException, CustomerIdNotFoundException {
    Customer customer = customerFinder.retrieveCustomer(customerId);
    if (customer.getCart().isEmpty())
        throw new EmptyCartException(customer.getName());
    Order newOrder = payment.payOrderFromCart(customer, cartPriceFromCustomer(customer));
    customer.clearCart();
    return newOrder;
}
```






```java
@Component
public class Cashier implements Payment {

    private final Bank bank;

    private final OrderCreator orderer;

    private final OrderCooking kitchen;

    public Cashier(Bank bank, OrderCreator orderer, OrderCooking orderCooking) {
        this.bank = bank;
        this.orderer = orderer;
        this.kitchen = orderCooking;
    }

    @Override
    @Transactional(propagation = Propagation.MANDATORY)
    public Order payOrderFromCart(Customer customer, double price) throws PaymentException {
        String paymentReceiptId  = bank.pay(customer, price).orElseThrow(() -> new PaymentException(customer.getName(), price));
        Order order = orderer.createOrder(customer, price, paymentReceiptId);
        return kitchen.processInKitchen(order);
    }

}
```

Business object only



Internal component (no controller)
+ connector to external services

```java
@Component
public class BankProxy implements Bank {

    private static final Logger LOG = LoggerFactory.getLogger(BankProxy.class);

    private final String bankHostandPort;

    private final RestClient restClient;

    public BankProxy(@Value("${bank.host.baseurl}") String bankHostandPort, RestClient.Builder restClientBuilder) {
        this.bankHostandPort = bankHostandPort;
        this.restClient = restClientBuilder.baseUrl(this.bankHostandPort).build();
    }

    @Override
    @Transactional(propagation = Propagation.MANDATORY)
    public Optional<String> pay(Customer customer, double value) {
        try {
            ResponseEntity<PaymentReceiptDTO> responseEntity = restClient.post()
                    .uri("/cctransactions")
                    .contentType(MediaType.APPLICATION_JSON)
                    .body(new PaymentRequestDTO(customer.getCreditCard(), value))
                    .retrieve()
                    .toEntity(PaymentReceiptDTO.class);
```

https://github.com/CookieFactoryInSpring/simpleTCFS

chapters / BusinessComponents.md

Answers

Business Components

- Author: Philippe Collet

We focus here on the implementation of a first component, dedicated to handle customer's carts, enables to show the main mechanisms of the business component layer.

Provided Interfaces

The component is very basic, still we apply interface segregation with two interfaces:

- `CartModifier` : operations to modify a given customer's cart, like adding or removing cookies, and to retrieve the contents of the cart;

```
public interface CartModifier {

    Item update(Long customerId, Item it) throws NegativeQuantityException, CustomerIdNotFoundExce

    Set<Item> cartContent(Long customerId) throws CustomerIdNotFoundException;
}
```

- `CartProcessor` : operations for computing the cart's price and validating it to process the associated order;

```
public interface CartProcessor {
```

TCF MVP

Component Assembly?

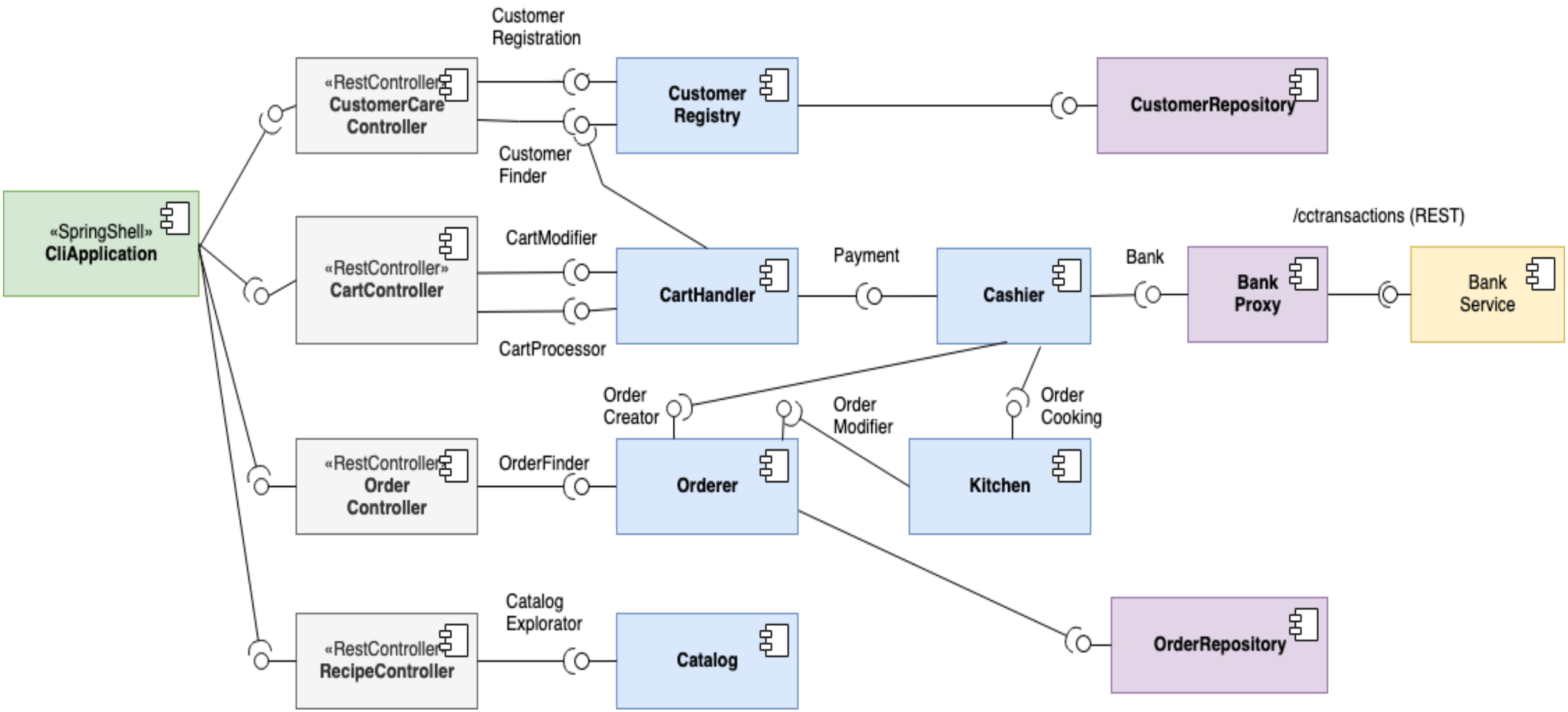
CLI (Springshell)
cli directory

Interoperability
REST API
(Spring Controller)

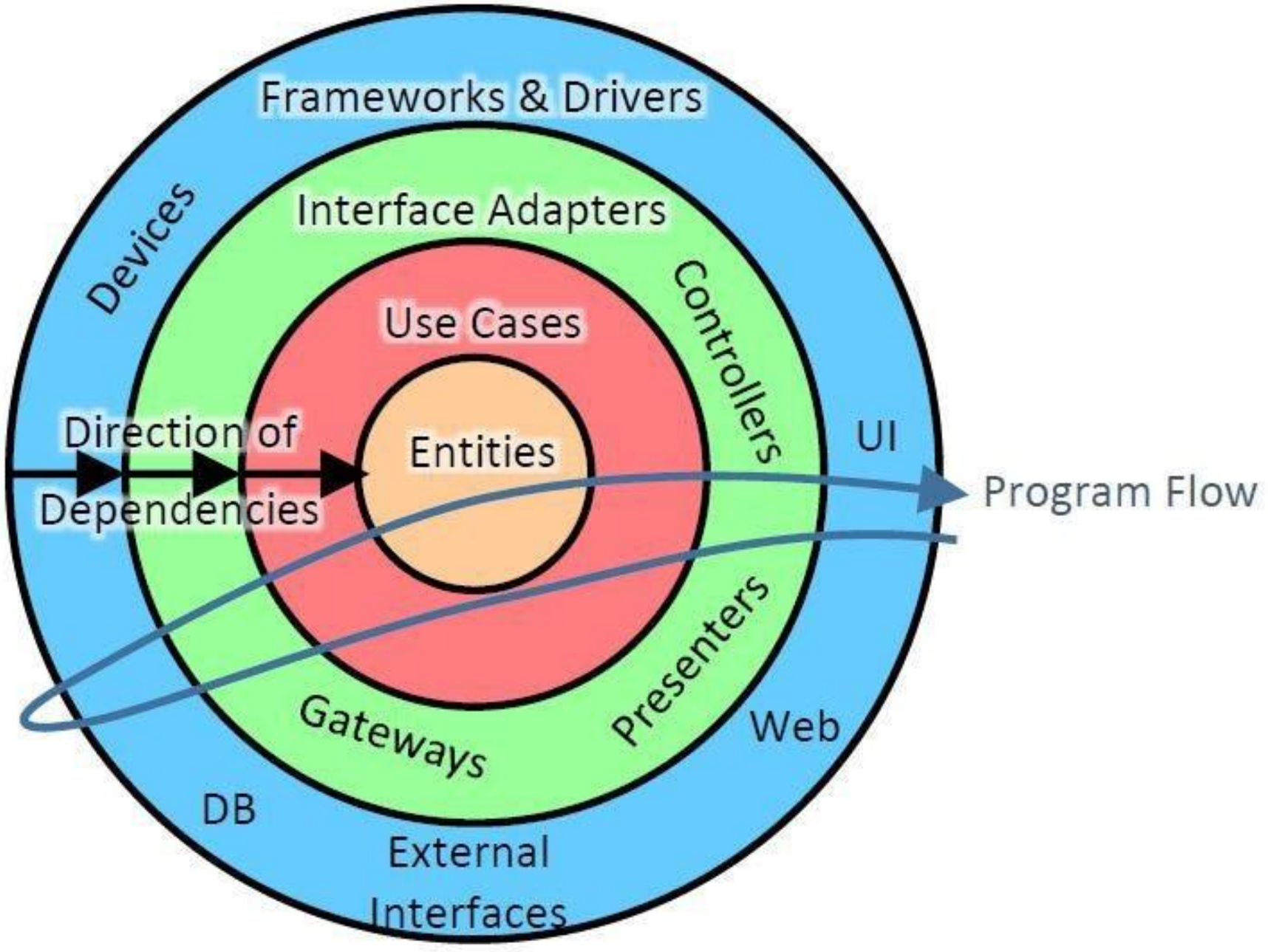
Spring Components

Spring Connectors &
Repositories

External service

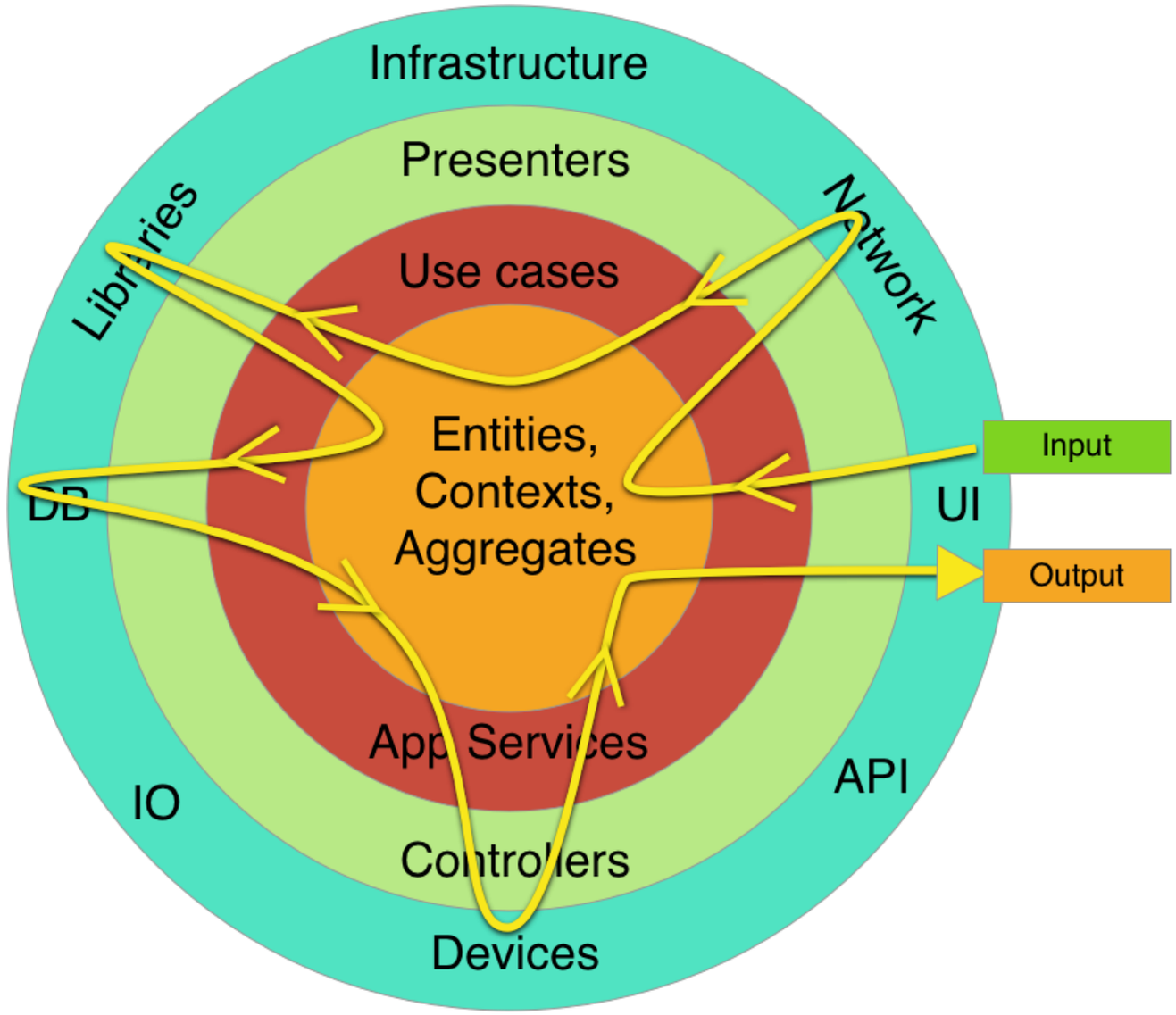


Clean Architecture & Program Flow



https://medium.com/@omid-ahmadpour/clean-architecture-template-with-net-and-its-importance-e5b3b97a6e48

Clean Architecture & Program Flow



https://crosp.net/blog/software-architecture/clean-architecture-part-2-the-clean-architecture/

TCF MVP

Controllers?

chapters / Controllers.md

Answers

REST controllers (and RestClient)

- Author: Philippe Collet

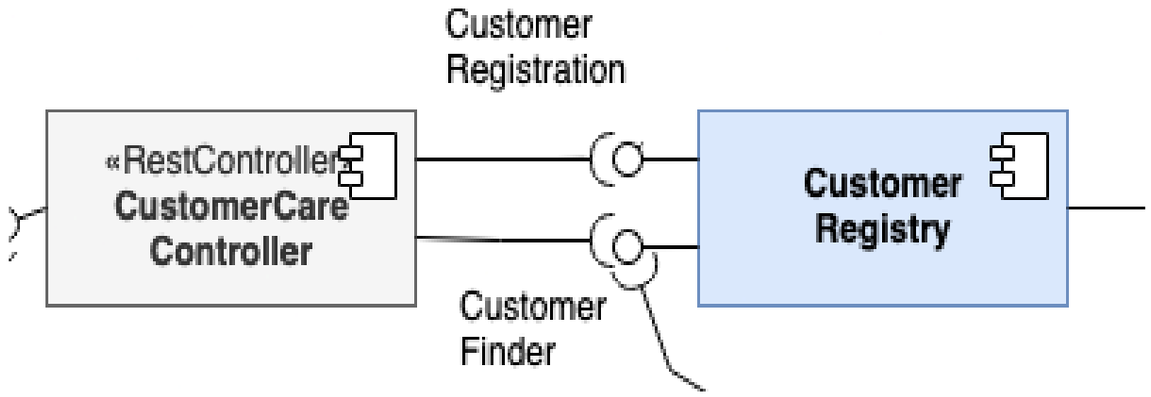
The aim of this chapter is to provide the basics about implementing RESTful services, as well as calling external REST services. Details about testing are available in the dedicated chapter.

To implement RESTful services, Spring can rely on two framework, a simple and synchronous Web model-view-controller (MVC) framework, and an asynchronous WebFlux framework. To ease understanding, we use here the MVC one, which was originally designed around a DispatcherServlet that dispatches requests to handlers, with configurable handler mappings, view resolution, as well as support for uploading files. It can be noted that thanks to virtual threads in Java 21, the synchronous model is regaining interest as it can now scale better. The default handler is based on the `@Controller` and `@RequestMapping annotations`, offering handling methods for the *http* protocol. Afterwards the controller mechanism enables one to create RESTful servers with extended annotations.

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.

+ next lecture!



Specific type of component

Mapping to the REST route

```
@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;

public CustomerCareController(CustomerRegistration registry, CustomerFinder finder) {
    this.registry = registry;
    this.finder = finder;
}
```

Injection (as usual) to connect to the business layer and delegate business logic

A method to implement the route + the http verb (POST here)

```
@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, exposing
        return ResponseEntity.status(HttpStatus.CONFLICT).build();
    }
}
```

DTO with conversion util methods

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

TCF MVP

Database?

Persistence?

chapters / Persistence.md

Answers

On Persistence

- Author: Philippe Collet

We focus here on the setup of the persistence layer using:

- a dockerized Postgresql DB (as a real relational database)
- a H2 in-memory database for testing (which is the default setup in Spring JPA for testing)

Configuration

We have to first extend the pom.xml:

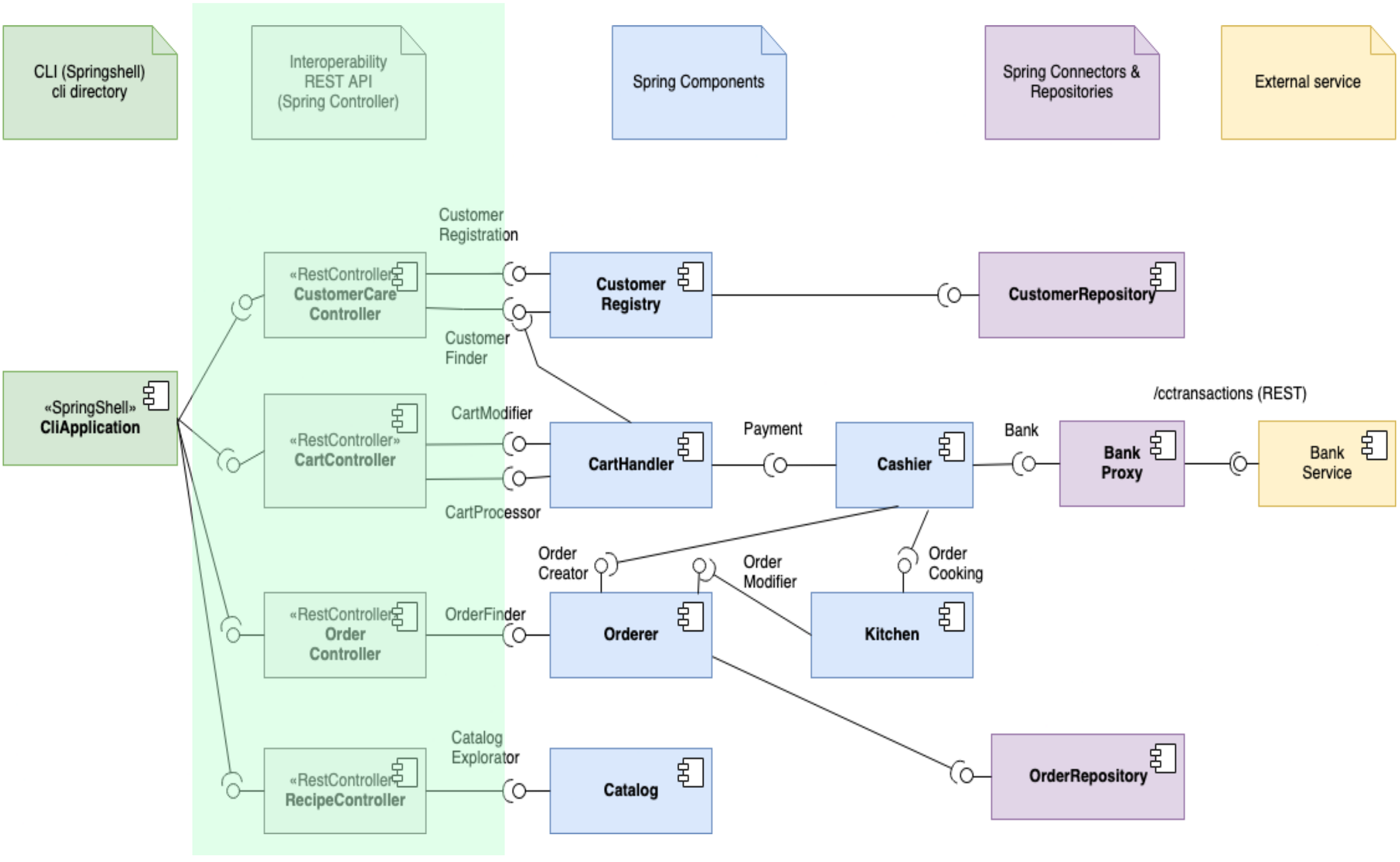
```
        <dependency>
            <groupId>org.springframework.boot</groupId> <!-- JPA + hibernate-core default support -->
            <artifactId>spring-boot-starter-data-jpa</artifactId>
        </dependency>
        <dependency>
            <groupId>com.h2database</groupId>
            <artifactId>h2</artifactId>
```

+ lecture in few weeks

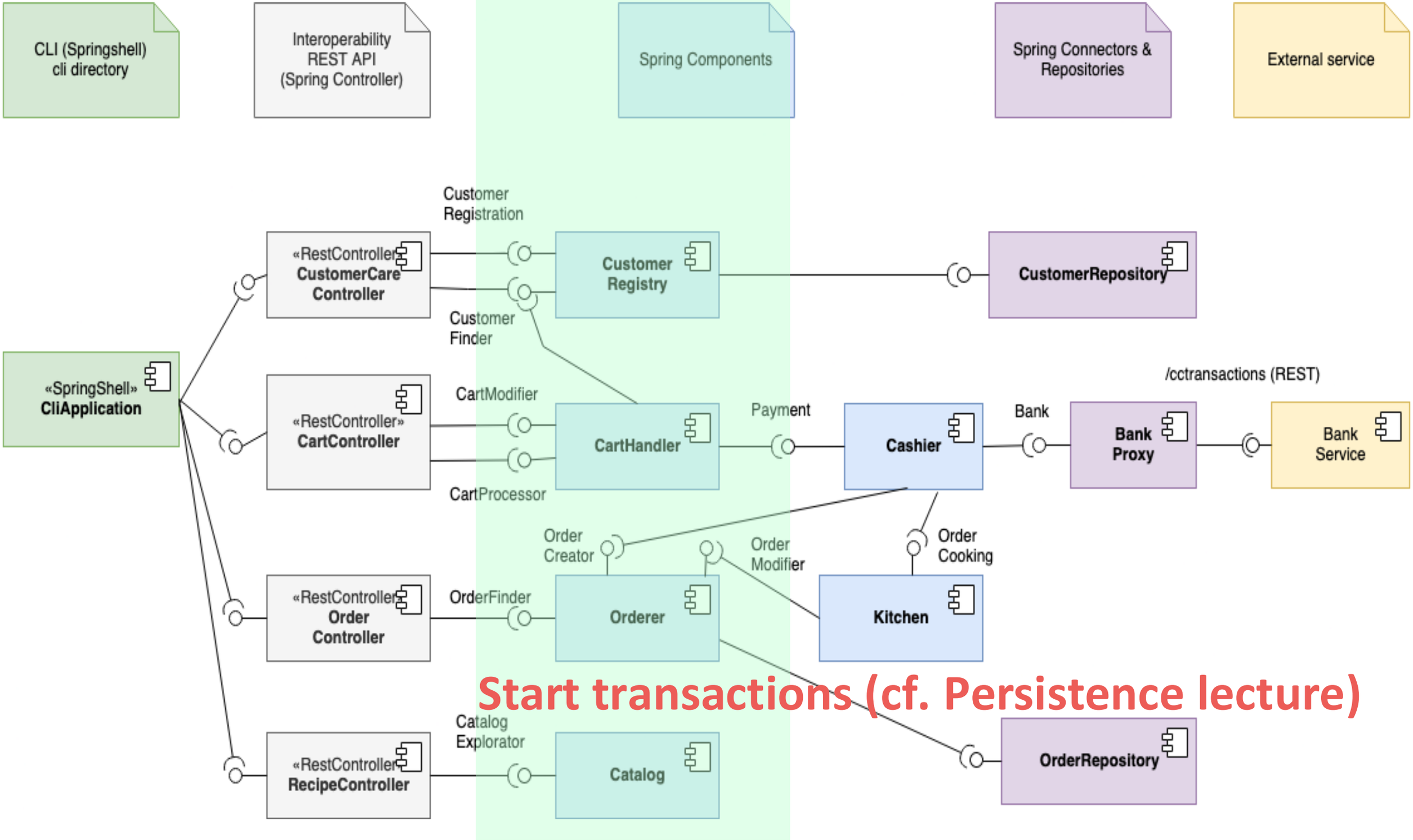
MVP

Layers ?

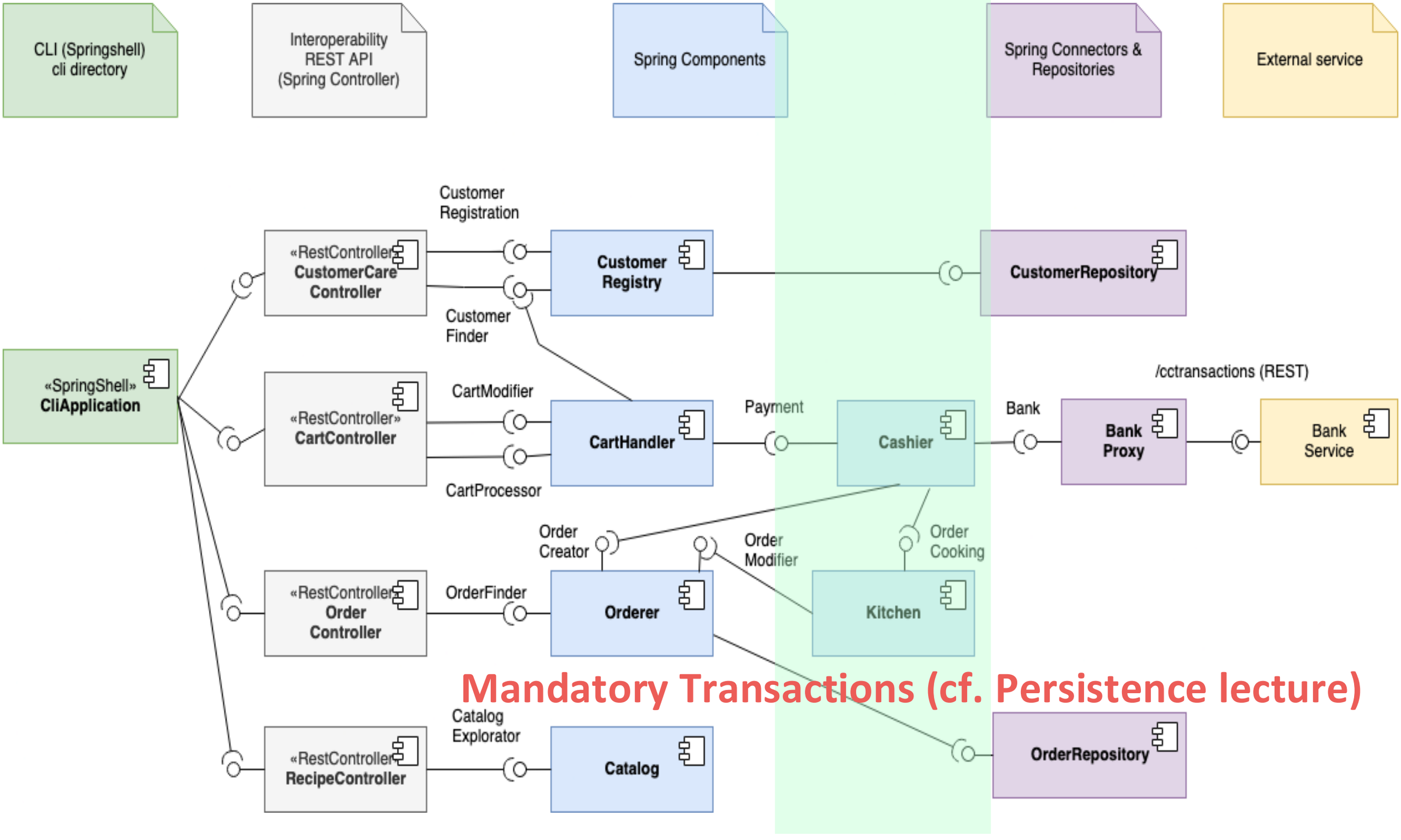
Controllers: DTO (Data Transfert Object) with encoding/decoding + raw validation of input (no business logic)



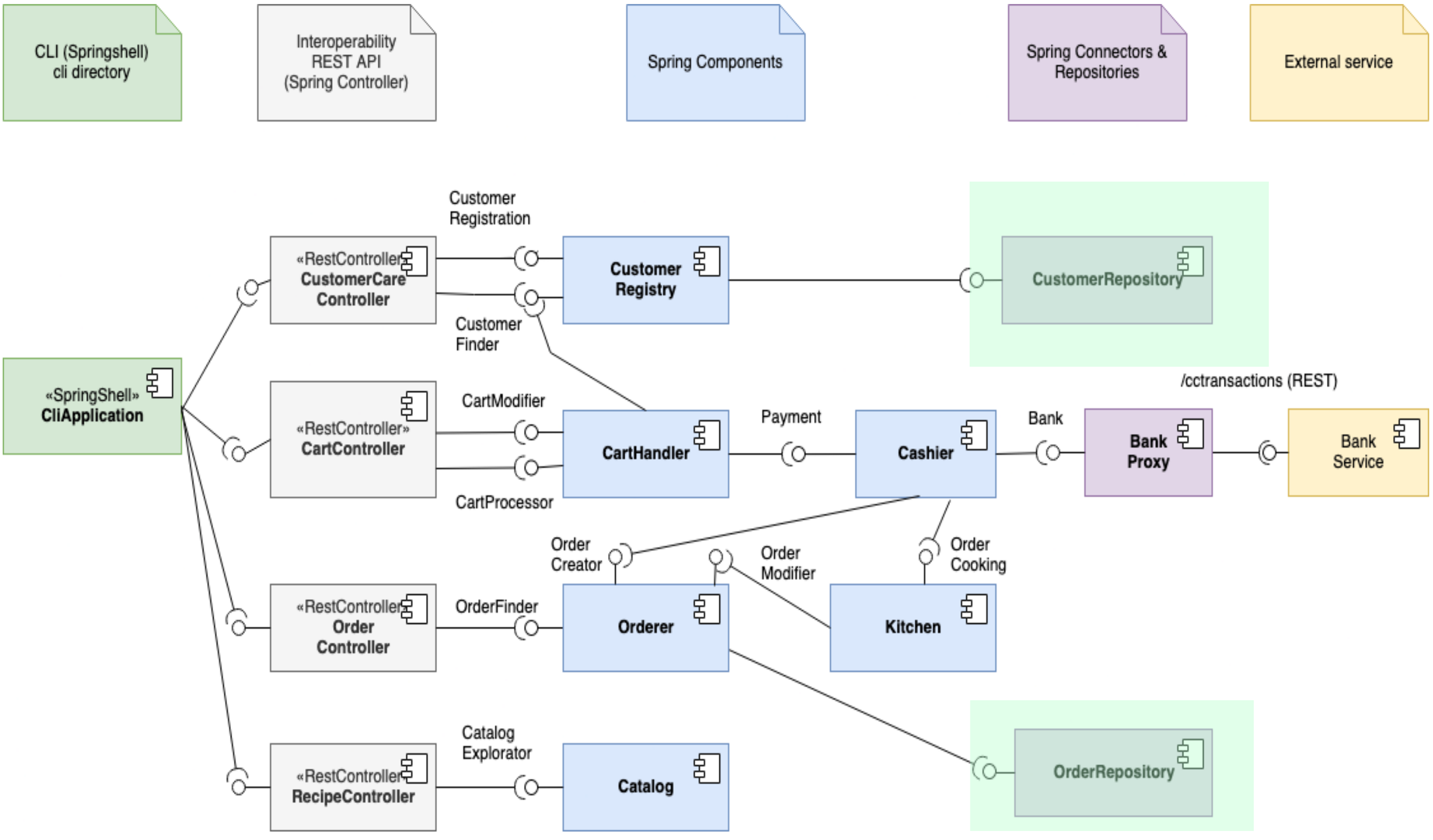
Service (component): Facade of the business logic – user visible actions
DTO or Id as input parameters and output values



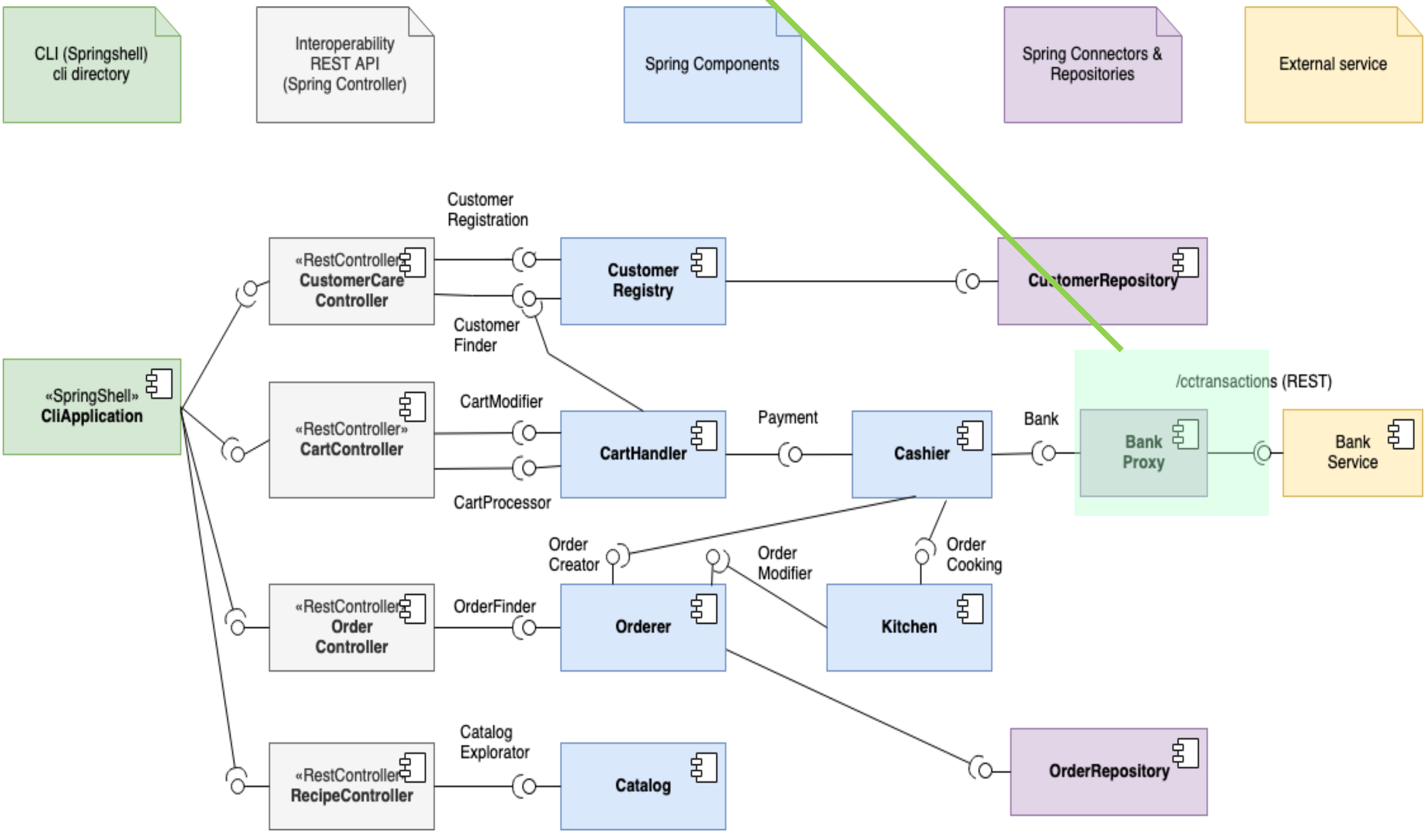
Business components: Internal components of the business logic – Decomposition of internal behaviors – objects as parameters/results



Repositories: technical facade to the data source (the DB or whatever is used as a source to retrieve and/or save data)



Connectors: Proxy interfaces to external systems (with DTO as input/output)



Where is the core business logic?

CLI (Springshell)
cli directory

Interoperability
REST API
(Spring Controller)

Spring Components

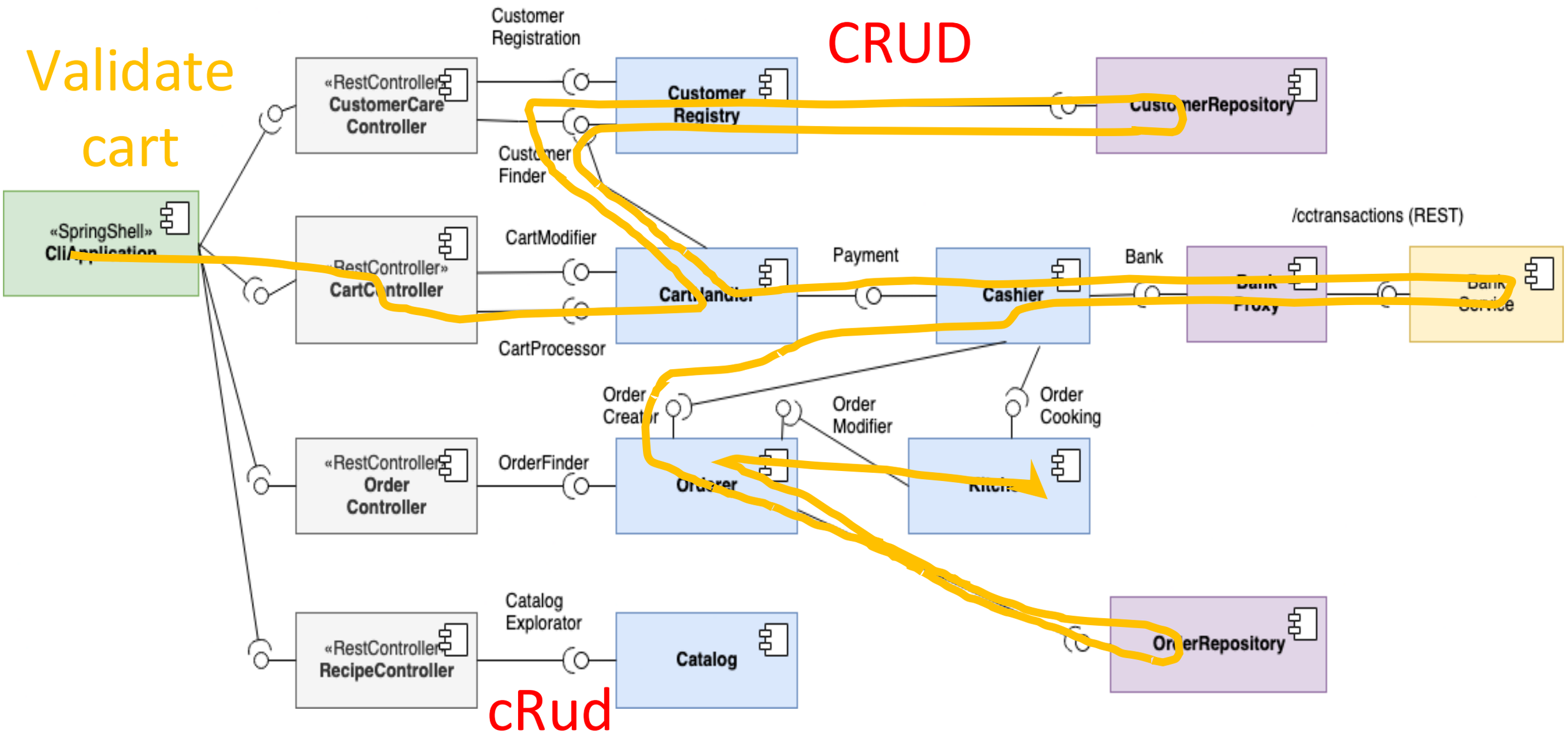
Spring Connectors &
Repositories

External service

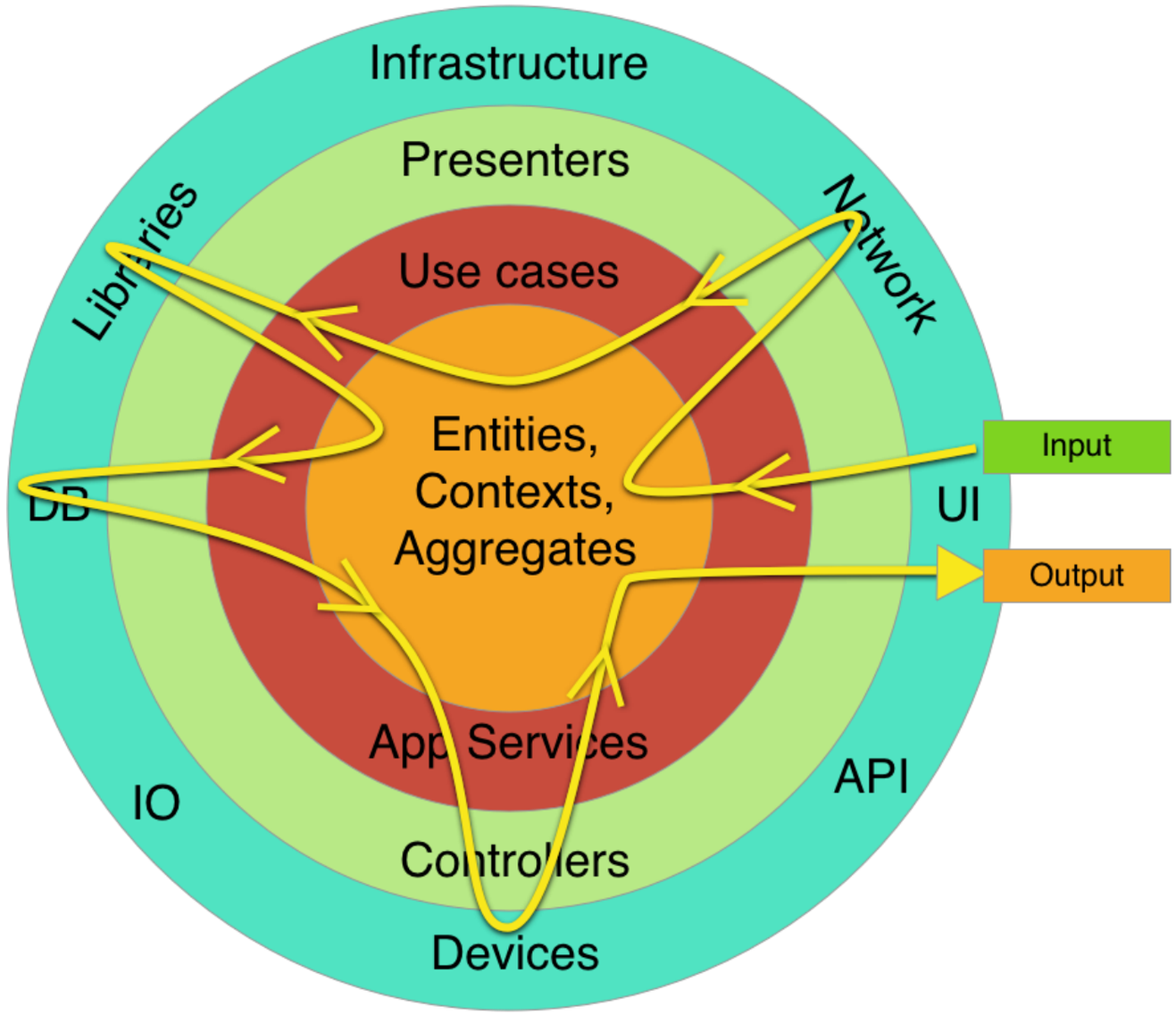
Validate
cart

CRUD

cRud

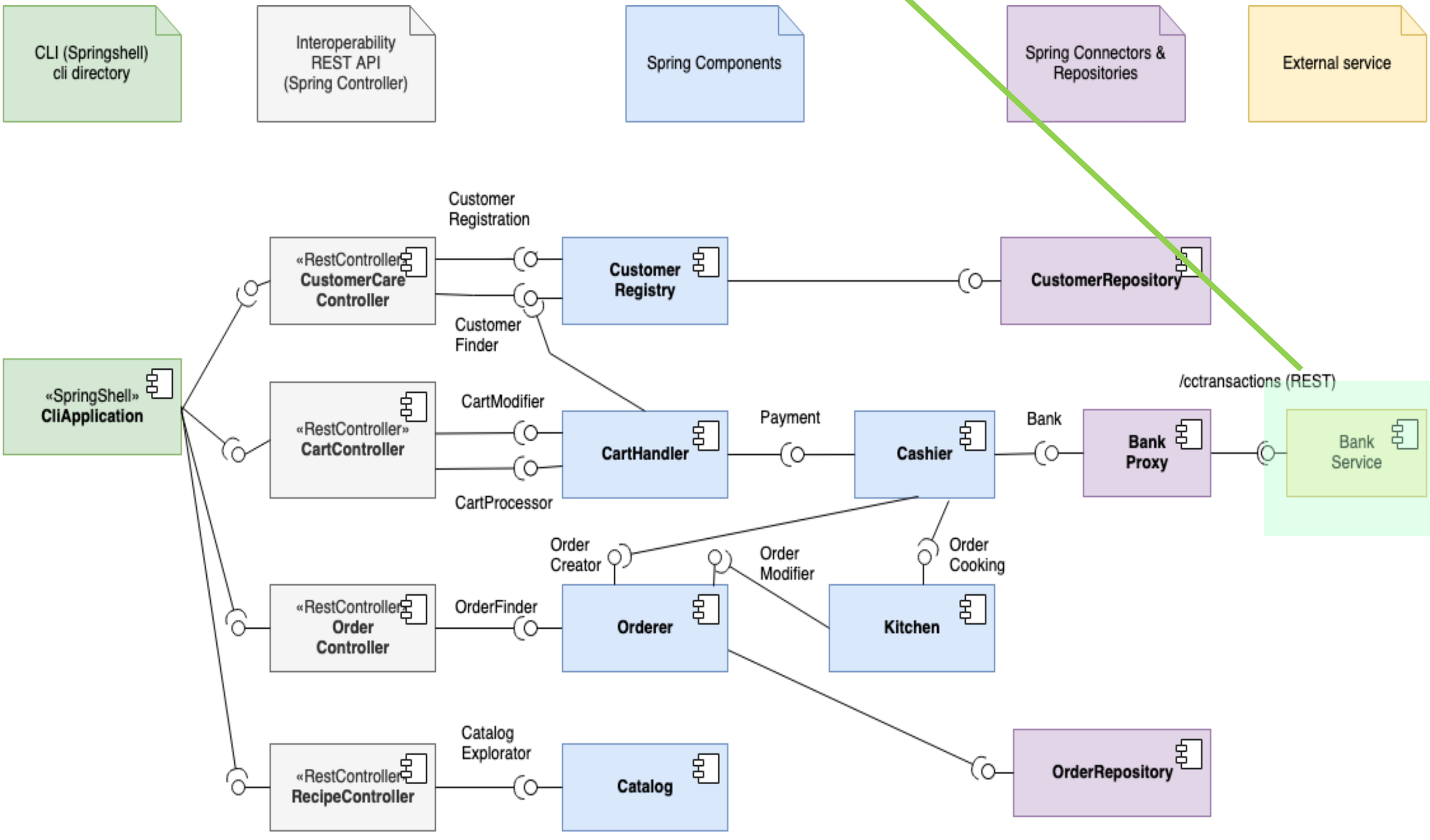


Clean Architecture & Program Flow!!!!!!!!!

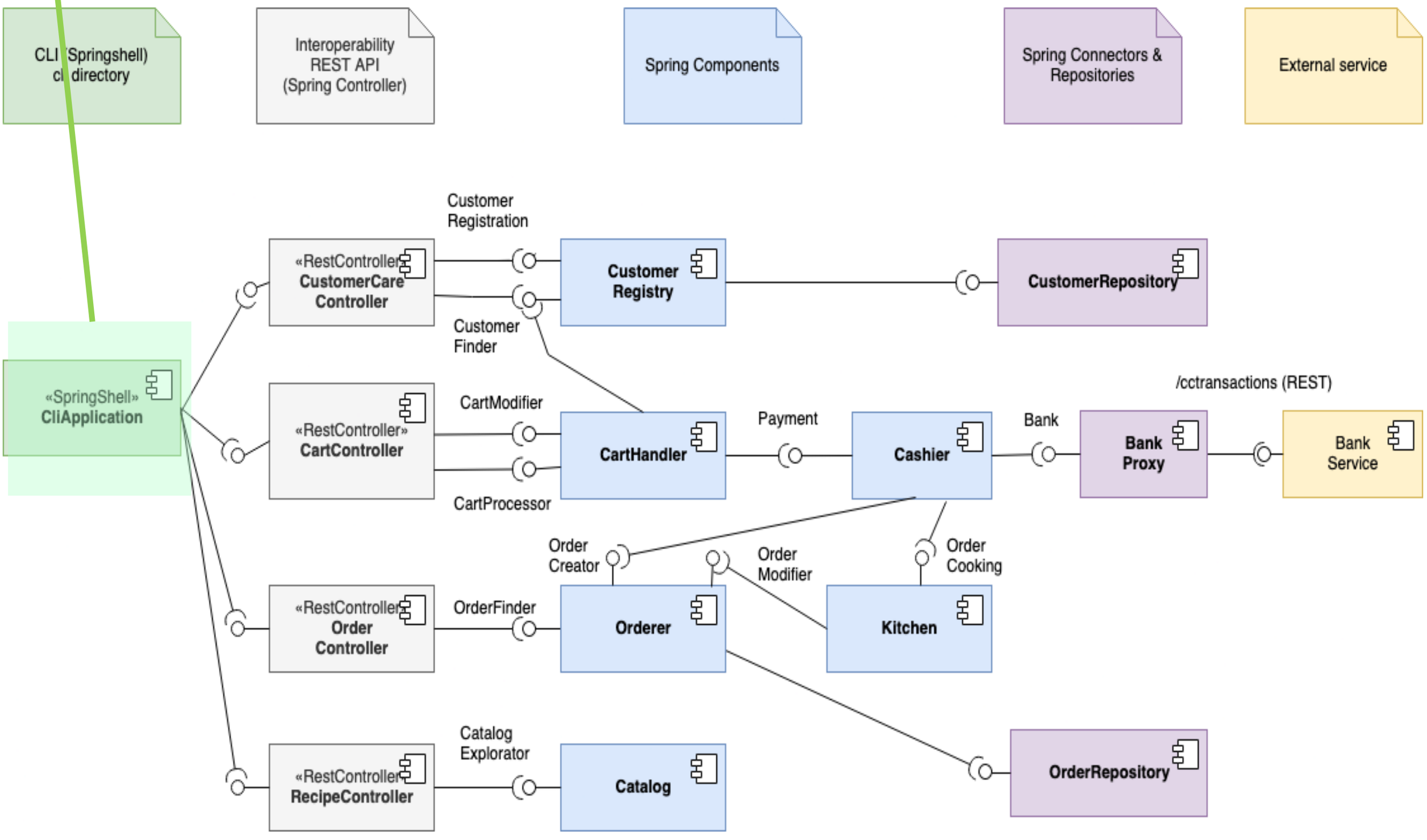


https://crosp.net/blog/software-architecture/clean-architecture-part-2-the-clean-architecture/

BankService : Ultra MVP in NestJS (dockerized)



CLI (Command Line Interface) : Spring shell (dockerized)



TCF MVP

How to deploy TCF?

Answers in the main README

To start and to run your stack

Everything containerized

While the details and separate build of each subsystem can be found in the following section, we run here the Spring backend with postgres, the CLI, and the external system into docker. It requires to build the three images while the `docker compose` will take care of retrieving an official postgres image and compose everything from the `compose.yaml` configuration.

To understand how it works inside

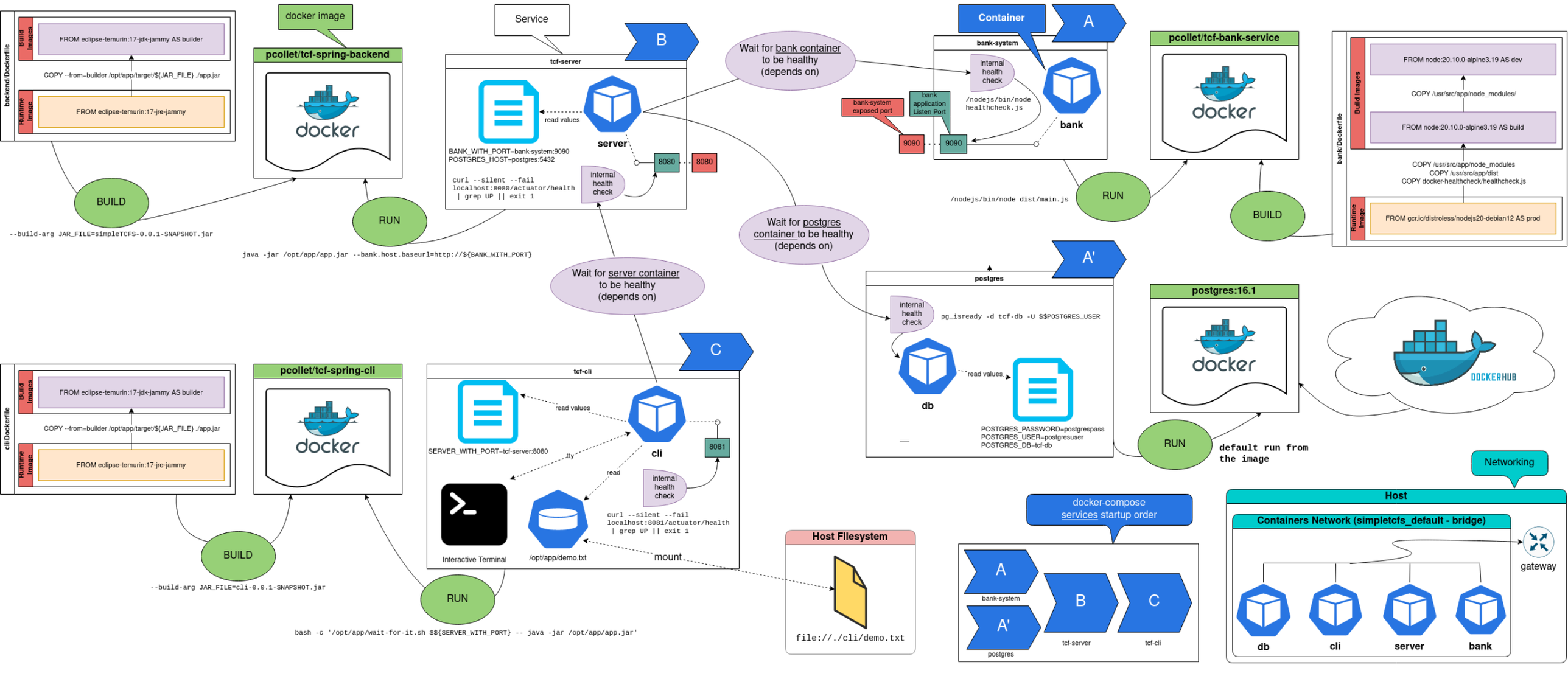
Basic build and run

Outside docker, the first step is to build the backend and the cli. This can be done manually, from both folders (it will generate the corresponding jar into the target folder), using the command:

Answers in the Architecture chapter

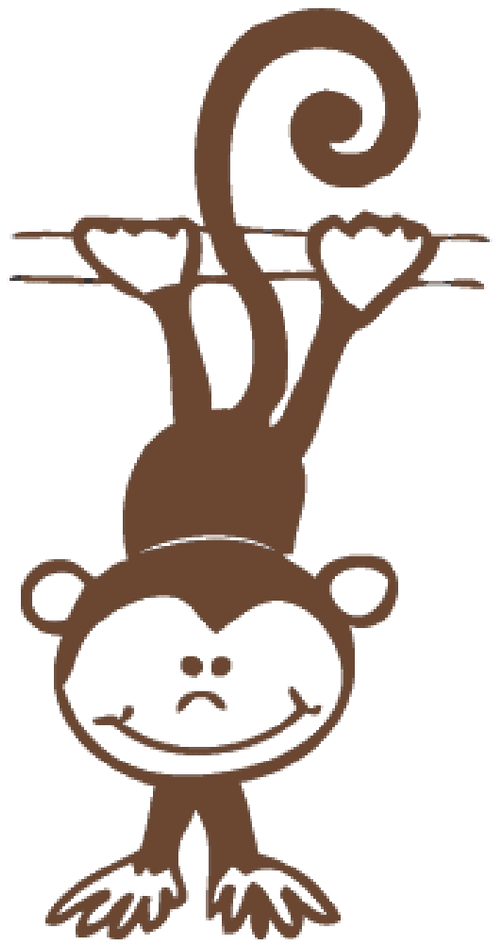
Docker composition schema

To ease understanding of the docker composition provided, the following schema illustrates all main elements:



https://github.com/CookieFactoryInSpring/simpleTCFS

monkey see



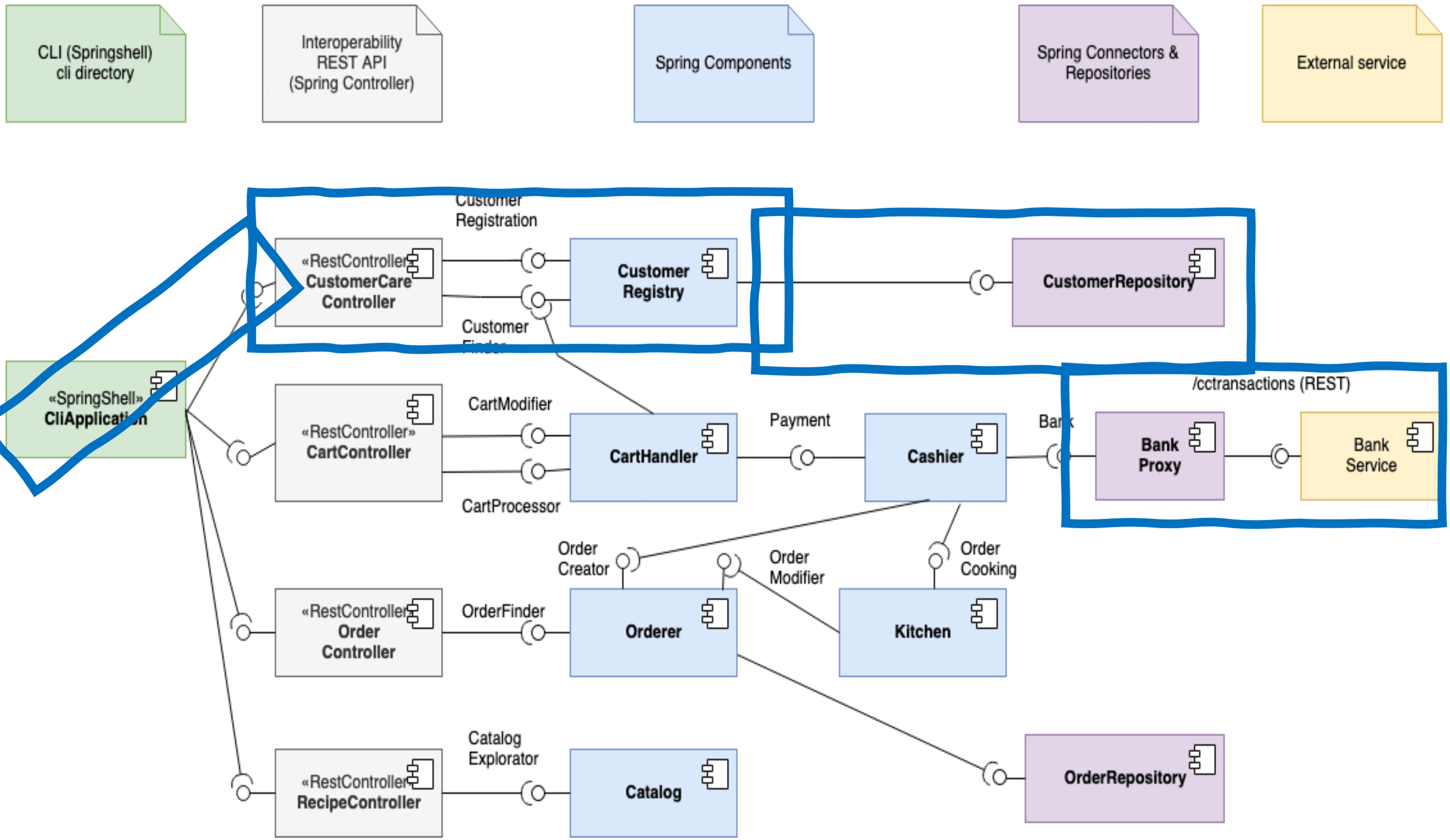
monkey do

TCF MVP

How to reuse TCF?

1. Understand TCF

2. Extract a useful part



Contraintes de développement (liées à DevOps)

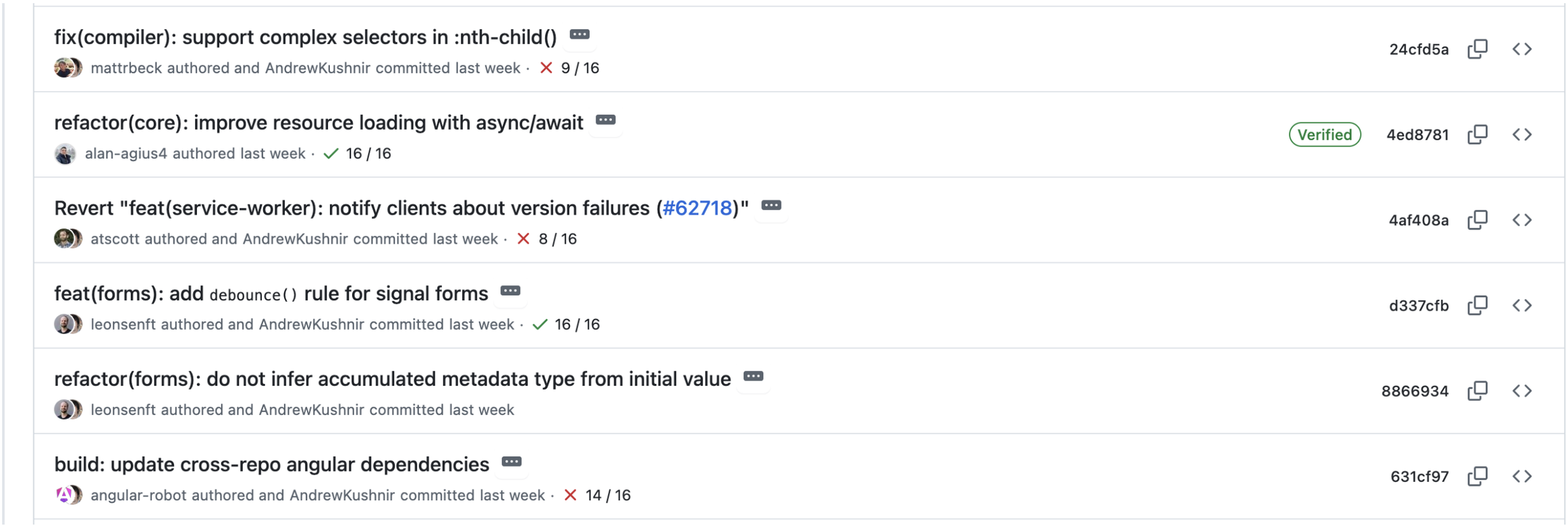
Au moins un fonctionnement git avec une branche dev (instable) et main (stable)

Tous les commits suivent le f

Conventional Commits

A specification for adding human and machine readable meaning to commit messages

https://www.conventionalcommits.org/en/v1.0.0/



Tous les détails juste après la livraison du rapport d’architecture

Contraintes techniques

Le projet (MVP de l'étape 1 y compris) développé doit reprendre exactement la pile technologie fournie par la CookieFactory (https://github.com/CookieFactoryInSpring/simpleTCFS) :

- Une ou plusieurs « cli » de pilotage de la démo
- Un backend en SpringBoot reprenant le setup (pom.xml, organisation du code, dépendances identiques à la version près) fourni
- Des services externes simplifiés ou mockés implémentés avec l'architecture NestJS de la *bank* fournie dans la CookieFactory ou l'architecture SpringBoot du backend.

Toute variation sera considérée comme une faute grave et pénalisera lourdement l'évaluation.

Contraintes sur le framework SpringBoot

- **Utiliser le setup CookieFactory**
- **Ne pas ajouter d'autres frameworks, par exemple :**
  - Pas de Lombok
  - Pas de MapStruct