实验报告：使用SSM框架实现客户表的增删改查功能

本实验报告旨在展示如何使用SSM（Spring, SpringMVC, MyBatis）框架实现客户表的增删改查（CRUD）功能，首先，确保MySQL数据库已启动，并创建名为ecom的数据库。然后执行以下SQL语句创建customers表：

|  |
| --- |
| create table customers  (  customer\_id binary(16) not null  primary key,  phone\_number varchar(128) null,  address varchar(128) null,  username varchar(128) not null,  password varchar(128) not null,  balance decimal not null  ); |

## CustomersDAO.java类结构

|  |
| --- |
| @Mapper  public interface CustomersDAO {  @Insert("""  INSERT INTO customers (  customer\_id,  phone\_number,  address,  username,  password,  balance  ) VALUES (  #{customer\_id},  #{phone\_number},  #{address},  #{username},  #{password},  #{balance}  )  """)  void addCustomer(Customer customer);  @Update("""  UPDATE customers  SET phone\_number = #{phone\_number},  address = #{address},  username = #{username},  password = #{password},  balance = #{balance}  WHERE customer\_id = #{customer\_id}  """)  void updateCustomer(Customer customer);  @Update("""  UPDATE customers  SET phone\_number = #{phone\_number},  address = #{address},  username = #{username},  balance = #{balance}  WHERE customer\_id = #{customer\_id}  """)  void updateCustomerNoPassword(Customer customer);  @Delete("""  DELETE FROM customers  WHERE customer\_id = #{customer\_id}  """)  void deleteCustomer(Customer customer);  @Select("""  SELECT \*  FROM customers  WHERE customer\_id = #{id}  """)  Customer getCustomer(UUID id);  @Select("""  SELECT \*  FROM customers  WHERE username = #{username}  """)  Customer getCustomerByName(String username);  @Select("""  SELECT \*  FROM customers  """)  List<Customer> getAll();  } |

CustomersDAO 接口是一个 MyBatis Mapper 接口，它定义了与客户数据交互的持久层操作。该接口通过注解方式将 Java 方法映射到 SQL 语句。addCustomer 方法用于向 customers 表中插入新的客户记录，包括客户ID、电话号码、地址、用户名、密码和余额。updateCustomer 方法根据客户ID更新客户的所有信息，包括电话号码、地址、用户名、密码和余额。updateCustomerNoPassword 方法与 updateCustomer 类似，但它在更新时不修改客户的密码，这在某些场景下，如用户只更新个人信息而不更改密码时非常有用。deleteCustomer 方法根据客户ID从数据库中删除客户记录。getCustomer 方法通过客户ID查询并返回单个客户对象。getCustomerByName 方法则允许通过客户的用户名查询客户信息。最后，getAll 方法用于检索 customers 表中的所有客户记录，并以列表形式返回。

## CustomerService.java 类结构

|  |
| --- |
| public interface CustomersService {  Customer addCustomer(final Customer customer);  Customer updateCustomer(final Customer customer);  void deleteCustomer(final Customer customer);  Customer getCustomer(final UUID customerId);  List<Customer> getCustomers();  List<Commodity> getCommoditiesByCustomerId(UUID customerId);  } |

CustomersService 接口定义了对客户信息进行业务逻辑操作的方法。它提供了添加、更新、删除和查询客户的功能，以及获取客户相关商品信息的能力。addCustomer 方法用于添加一个新的客户，它接收一个 Customer 对象作为参数并返回已添加的 Customer 对象。updateCustomer 方法用于更新现有客户的信息，它接收一个 Customer 对象作为参数并返回已更新的 Customer 对象。deleteCustomer 方法用于删除一个客户，它接收一个 Customer 对象作为参数，但没有返回值。getCustomer 方法用于根据客户ID获取单个客户的信息，它接收一个 UUID 类型的 customerId 作为参数并返回对应的 Customer 对象。getCustomers 方法用于获取所有客户的列表，它没有参数，返回一个 Customer 对象的列表。getCommoditiesByCustomerId 方法用于根据客户ID获取该客户购买的商品列表，它接收一个 UUID 类型的 customerId 作为参数并返回一个 Commodity 对象的列表。

## CustomerServiceImpl.java 类结构

|  |
| --- |
| @Service  @CacheConfig(cacheNames = "customers")  public class CustomersServiceImpl implements CustomersService {  private final CustomersDAO customersDAO;  private final CommoditiesDAO commoditiesDAO;  public CustomersServiceImpl(CustomersDAO customersDAO,  CommoditiesDAO commoditiesDAO) {  this.customersDAO = customersDAO;  this.commoditiesDAO = commoditiesDAO;  }  // --- Write Operations (Add, Update, Delete) ---  @Override  @CacheEvict(allEntries = true)  public Customer addCustomer(Customer customer) {  customersDAO.addCustomer(customer);  return customer;  }  @Override  @CachePut(key = "#customer.customer\_id")  public Customer updateCustomer(Customer customer) {  if (customer.password() == null  || customer.password().isEmpty()  || customer.password().isBlank())  customersDAO.updateCustomerNoPassword(customer);  else  customersDAO.updateCustomer(customer);  return customer;  }  @Override  @CacheEvict(  value = "customers",  allEntries = true,  beforeInvocation = true  )  public void deleteCustomer(Customer customer) {  customersDAO.deleteCustomer(customer);  }  // --- Read Operations (Get) ---  // These operations benefit from caching.  @Override  @Cacheable(key = "#customer\_id")  public Customer getCustomer(UUID customer\_id) {  System.out.println("Fetching customer from DB for ID: "  + customer\_id);  return customersDAO.getCustomer(customer\_id);  }  public Customer getCustomerByName(String name) {  System.out.println("Fetching customer from DB for Name: "  + name);  return customersDAO.getCustomerByName(name);  }  @Override  public List<Customer> getCustomers() {  System.out.println("Fetching all customers from DB");  return customersDAO.getAll();  }  @Override  public List<Commodity> getCommoditiesByCustomerId(UUID customerId) {  return commoditiesDAO.getCommoditiesByCustomerId(customerId);  }  } |

CustomersServiceImpl 类是 CustomersService 接口的实现，负责处理客户相关的业务逻辑，并集成了缓存机制以优化性能。该服务类通过构造函数注入 CustomersDAO 和 CommoditiesDAO 接口，以便与数据持久层进行交互。

对于写操作，如 addCustomer，在添加新客户后， @CacheEvict(allEntries = true) 注解会清除 customers 缓存中所有条目，确保数据一致性。updateCustomer 方法根据传入的 Customer 对象判断其密码是否为空、空字符串或只包含空格，如果密码为空，则调用 customersDAO.updateCustomerNoPassword 方法进行更新；否则，调用 customersDAO.updateCustomer 方法，并使用 @CachePut(key = "#customer.customer\_id") 注解更新缓存中对应客户ID的条目。deleteCustomer 方法在删除客户之前，使用 @CacheEvict(value = "customers", allEntries = true, beforeInvocation = true) 注解，强制清除所有 customers 缓存条目，以保证数据同步。

对于读操作，getCustomer 方法利用 @Cacheable(key = "#customer\_id") 注解，在首次调用时会从数据库获取客户信息并将其缓存，后续对相同 customer\_id 的请求将直接从缓存中获取，从而提高查询效率。getCustomerByName 方法根据客户名称从数据库中获取客户信息。getCustomers 方法则从数据库中获取所有客户的列表。此外，getCommoditiesByCustomerId 方法通过 commoditiesDAO 获取特定客户购买的商品列表。

## CustomerController.java

|  |
| --- |
| @RestController  @RequestMapping("/api/customer")  public class CustomerController {  private final CustomersServiceImpl customersService;  public CustomerController(CustomersServiceImpl customersService) {  this.customersService = customersService;  }  @GetMapping("customer")  public ResponseEntity<?> getCustomer(@RequestParam UUID id) {  return new ResponseEntity<>(customersService.getCustomer(id),  HttpStatus.OK);  }  @GetMapping("all/customers")  public ResponseEntity<?> getAllCustomers() {  return new ResponseEntity<>(customersService.getCustomers(),  HttpStatus.OK);  }  @PostMapping("customer")  public ResponseEntity<?> addCustomer(  @RequestParam String phone\_number,  @RequestParam String address,  @RequestParam String username,  @RequestParam String password) {  Customer customer = new Customer(phone\_number, address,  username, password);  customersService.addCustomer(customer);  return new ResponseEntity<>(HttpStatus.CREATED);  }  @GetMapping("commodities\_by\_customer")  public ResponseEntity<?> getCommoditiesByCustomerId(  @RequestParam UUID customerId) {  return new ResponseEntity<>(  customersService.getCommoditiesByCustomerId(  customerId),  HttpStatus.OK  );  }  @PutMapping("customer")  public ResponseEntity<?> updateCustomer(  @RequestBody Customer customer) {  return new ResponseEntity<>(  customersService.updateCustomer(customer),  HttpStatus.OK  );  }  @DeleteMapping("customer/{id}")  public ResponseEntity<?> deleteCustomer(@PathVariable UUID id) {  Customer customer = customersService.getCustomer(id);  if (customer == null) {  return new ResponseEntity<>("Customer not found",  HttpStatus.NOT\_FOUND);  }  customersService.deleteCustomer(customer);  return new ResponseEntity<>(HttpStatus.OK);  }  } |

CustomerController 是一个 Spring REST 控制器，负责处理客户相关的 HTTP 请求。它通过 @RestController 和 @RequestMapping("/api/customer") 注解暴露 RESTful API 接口，所有与客户相关的请求都将以 /api/customer作为基础路径。该控制器依赖于 CustomersServiceImpl 来执行实际的业务逻辑。

getCustomer 方法处理 GET 请求 /api/customer/customer，通过 @RequestParam UUID id 获取客户ID，并调用 customersService.getCustomer(id) 来检索特定客户的信息。如果成功，它会返回客户对象和 HttpStatus.OK。

getAllCustomers 方法处理 GET 请求 /api/customer/all/customers，用于获取所有客户的列表。它调用 customersService.getCustomers() 并返回所有客户的列表以及 HttpStatus.OK。

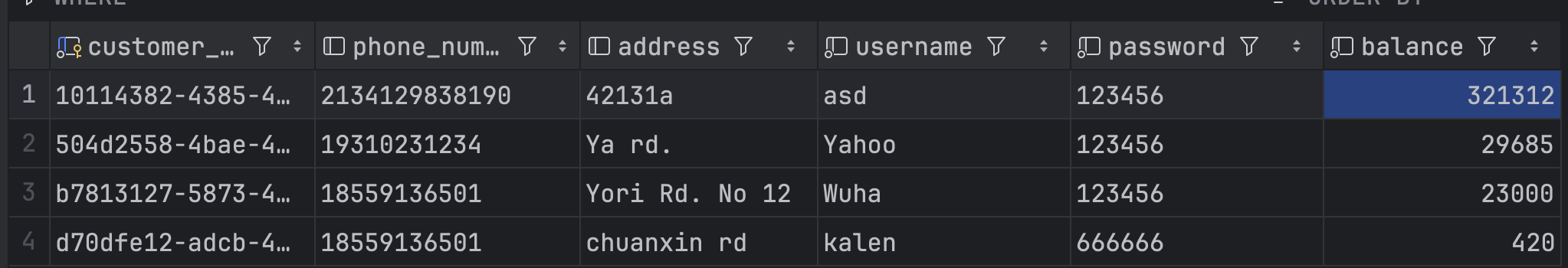
addCustomer 方法处理 POST 请求 /api/customer/customer，用于创建新客户。它通过 @RequestParam 获取 phone\_number、address、username 和 password，然后创建一个新的 Customer 对象并调用 customersService.addCustomer(customer) 进行添加。成功后返回 HttpStatus.CREATED。

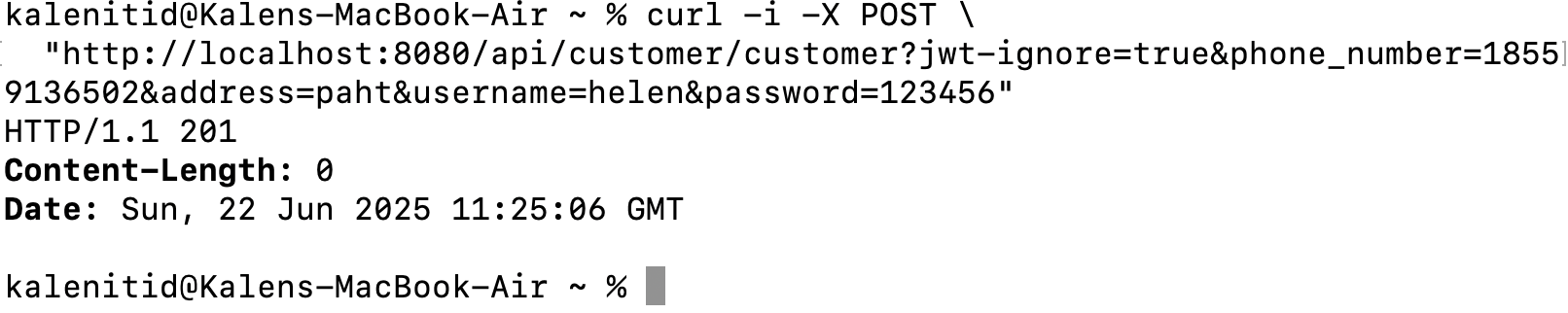
getCommoditiesByCustomerId 方法处理 GET 请求 /api/customer/commodities\_by\_customer，用于获取特定客户购买的商品列表。它通过 @RequestParam UUID customerId 获取客户ID，并调用 customersService.getCommoditiesByCustomerId(customerId)。成功后返回商品列表和 HttpStatus.OK。

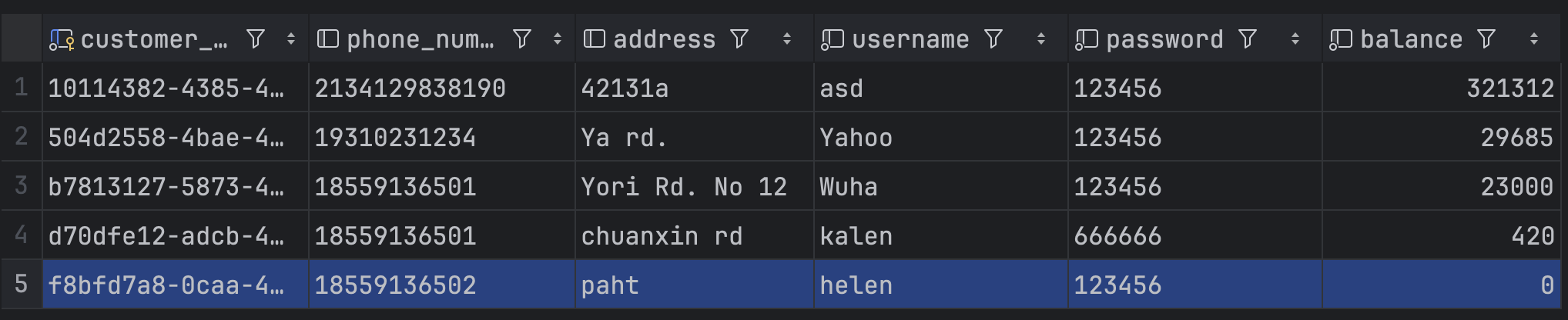
updateCustomer 方法处理 PUT 请求 /api/customer/customer，用于更新现有客户的信息。它通过 @RequestBody Customer customer 接收一个完整的 Customer 对象，并调用 customersService.updateCustomer(customer) 进行更新。成功后返回更新后的客户对象和 HttpStatus.OK。

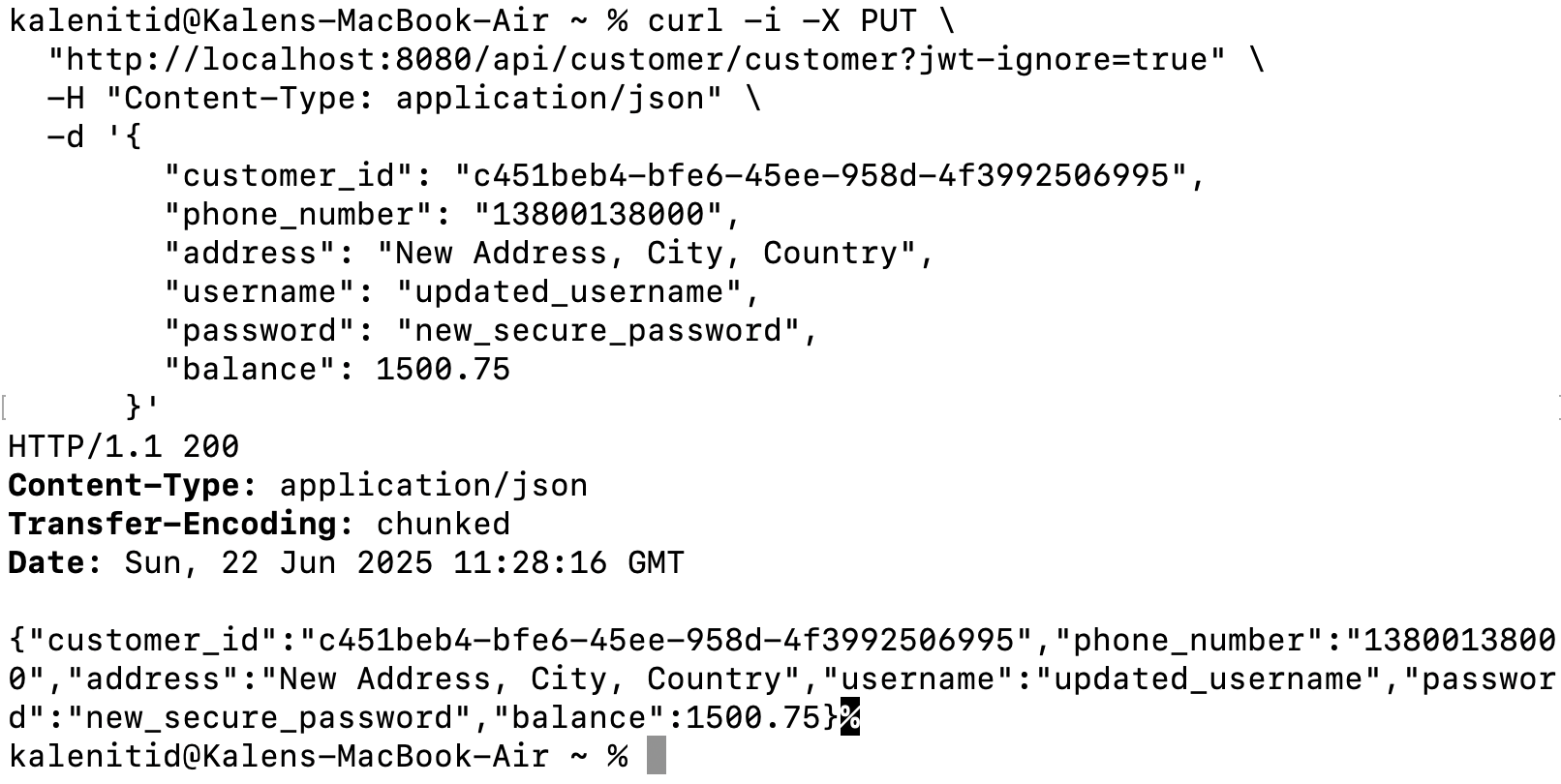
deleteCustomer 方法处理 DELETE 请求 /api/customer/customer/{id}，用于删除指定ID的客户。它通过 @PathVariable UUID id 获取客户ID。在删除之前，它会先调用 customersService.getCustomer(id) 检查客户是否存在。如果客户不存在，则返回 "Customer not found" 消息和 HttpStatus.NOT\_FOUND。如果客户存在，则调用 customersService.deleteCustomer(customer) 进行删除，并返回 HttpStatus.OK。

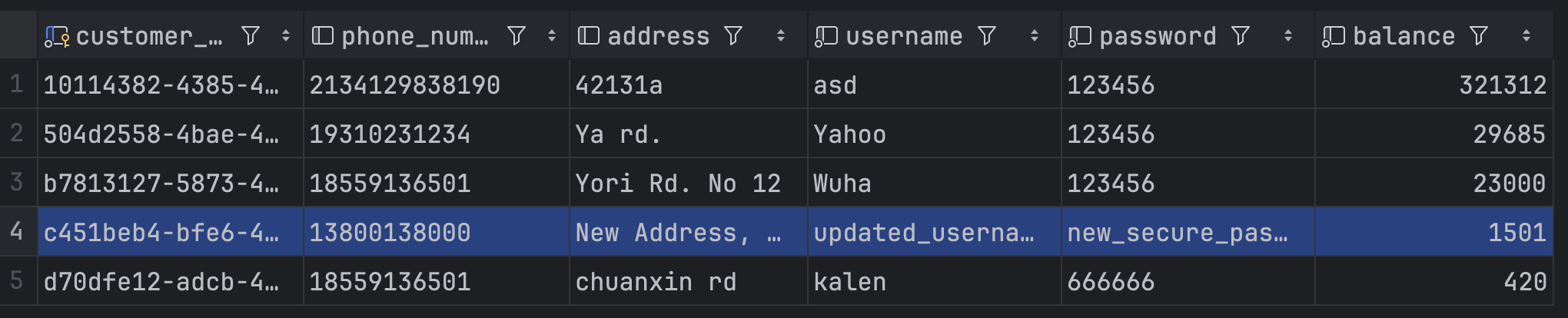
## 实验过程

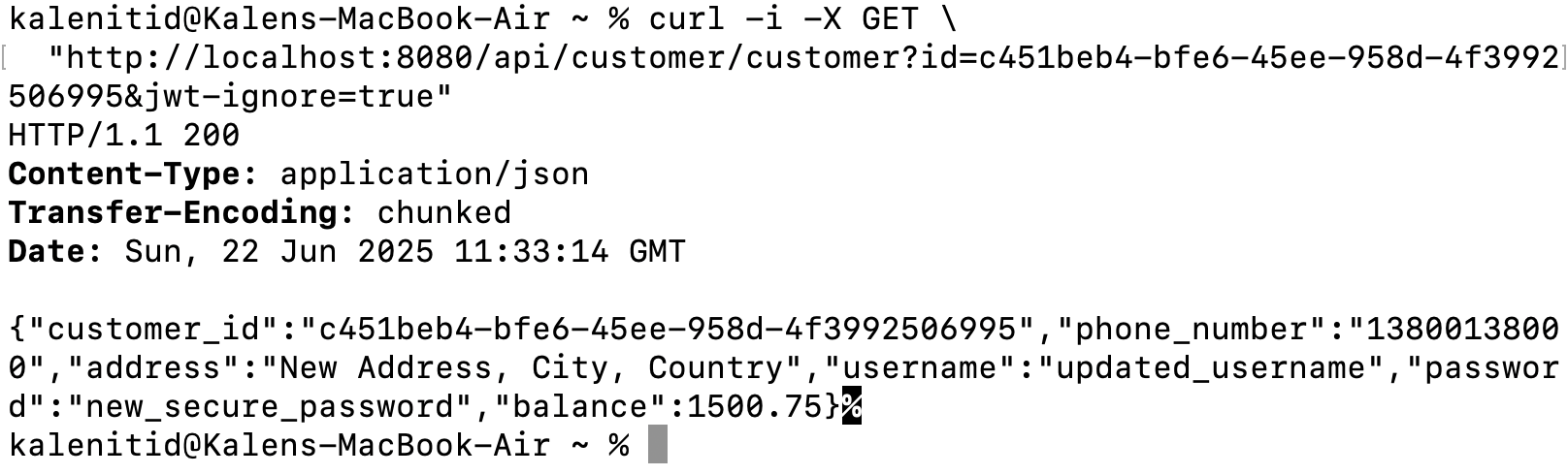
操作前的数据库状态：  


通过curl进行添加客户操作测试：  


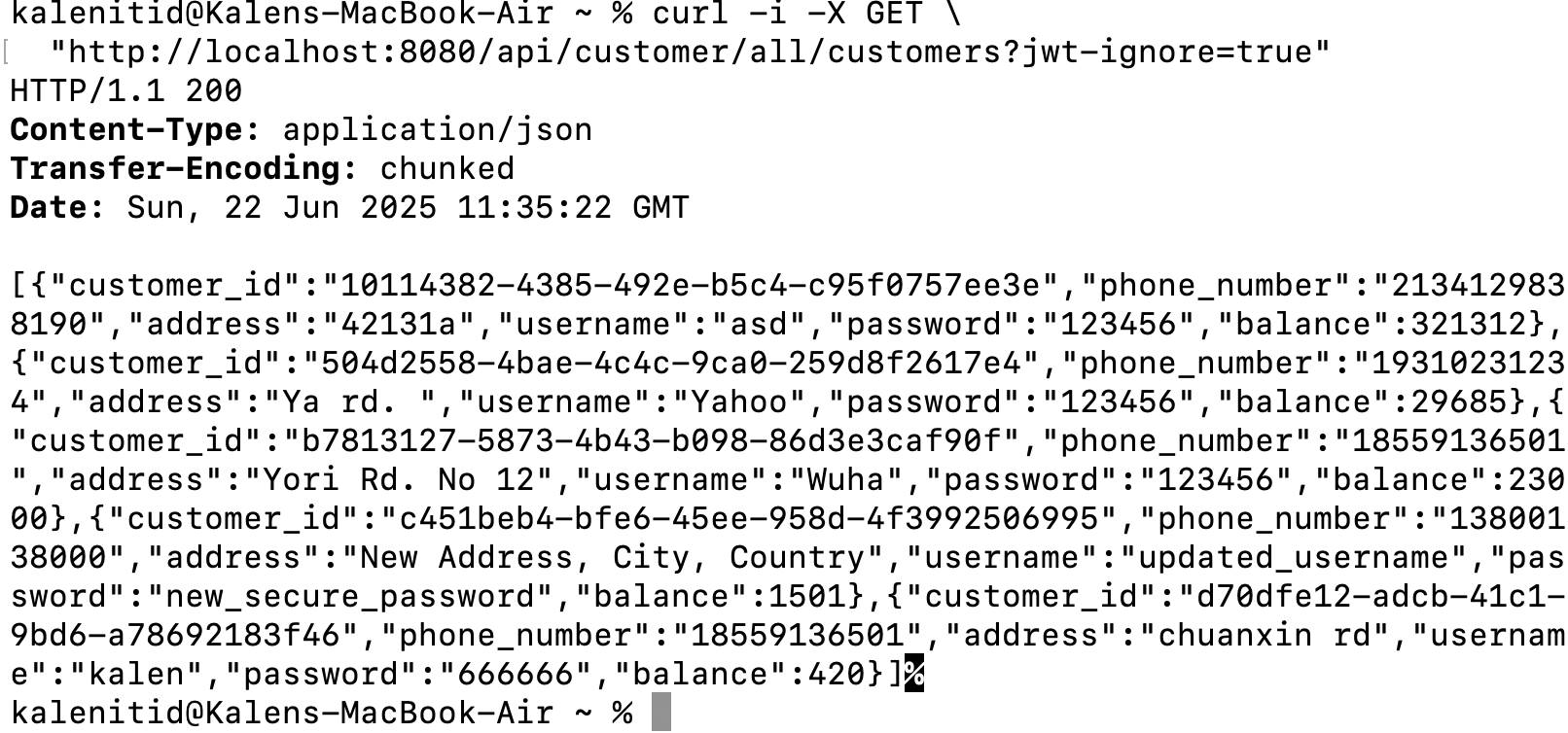
发出POST请求后，得到201状态码，代表客户创建成功。操作后的数据库状态如下图所示，名为helen的客户被成功加入数据库。  


通过curl进行客户信息修改操作：  


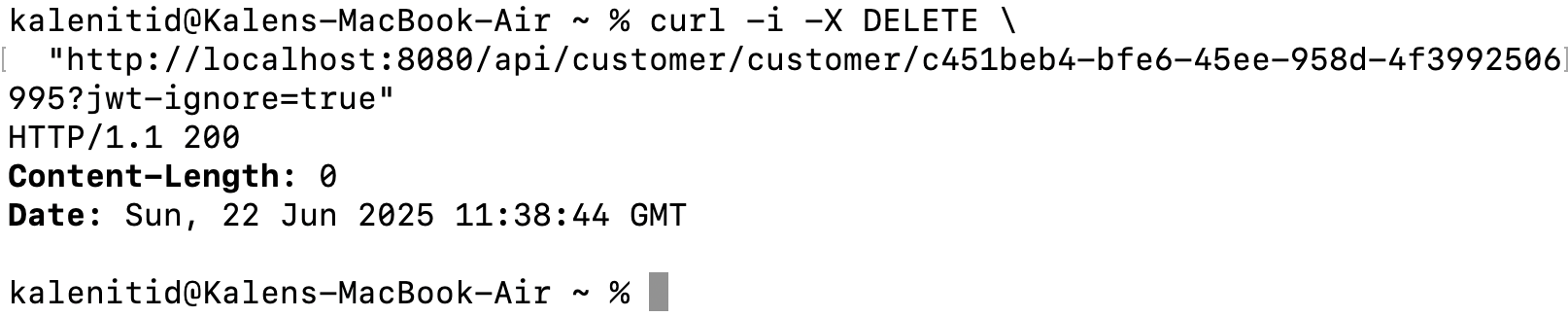
发出PUT请求后，得到200状态码，代表客户信息已经被成功修改。下面是成功修改客户后的数据库状态。（由于测试时曾重新添加客户，导致客户的UUID出现变动，实际上客户的UUID不会因为修改客户而改变）。  


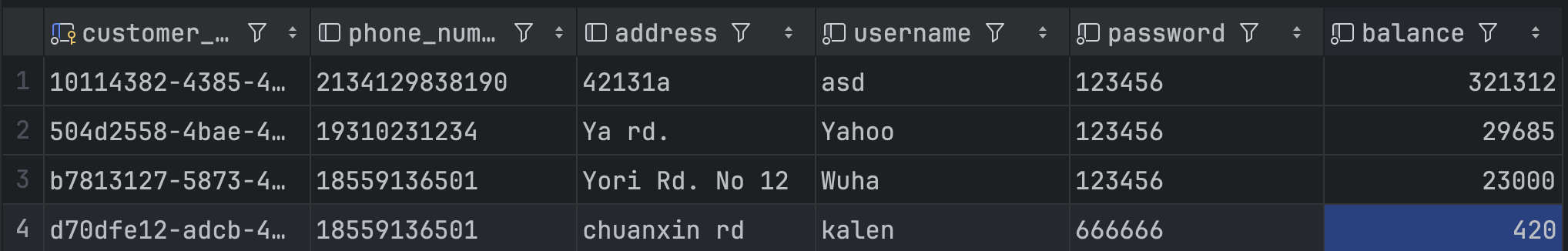
使用curl测试按uuid查询客户操作：  


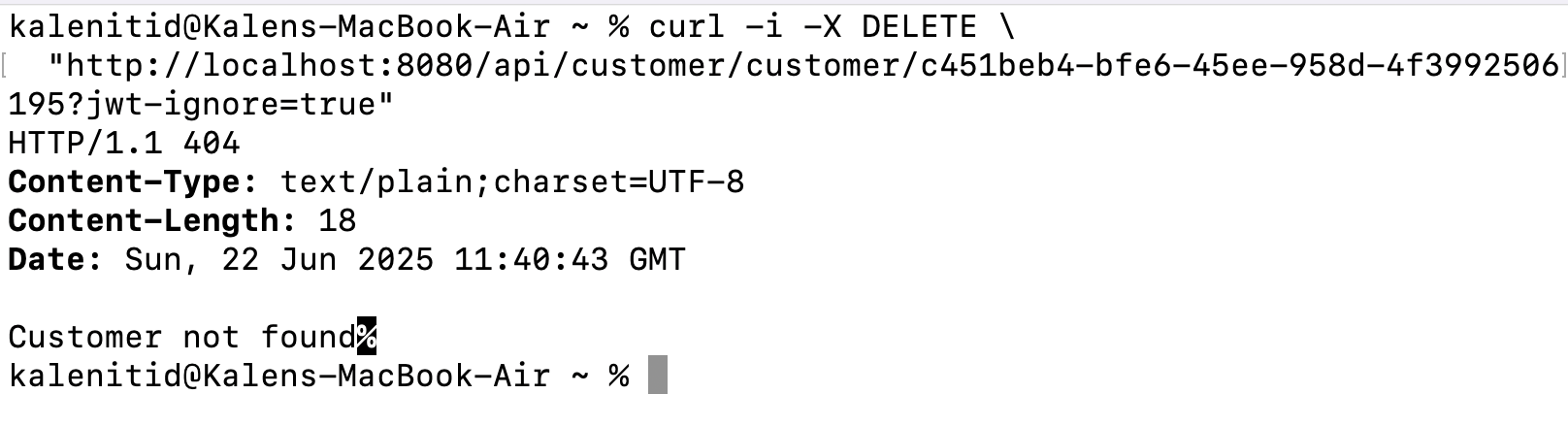
正确得到修改后的客户信息。

通过curl测试获取全部客户操作：  


与数据库中的项目相符，可见测试成功。

通过curl测试删除客户操作：  


执行删除操作后id所指客户成功被从数据库中删除  


测试使用curl删除不存在的客户，得到正确返回：  


## 总结

本系统构建了一个全面的客户管理模块，涵盖了从数据持久化到业务逻辑再到前端接口的完整功能。核心数据模型由 customers 表定义，该表存储了客户的唯一标识 customer\_id（以 BINARY(16) 形式存储）、电话号码、地址、用户名、密码和余额等信息 。Customer 类作为一个简单的POJO，映射了这些客户信息，并通过 uuidToBytes 和 bytesToUUID 辅助方法处理Java UUID 对象与MySQL BINARY(16) 数据类型之间的转换，确保了数据在不同层面的正确表示和持久化 。

在数据访问层 (DAO)，CustomersDAO 接口（在实验报告中由 RawJDBC.java 类实现具体操作）定义了对 customers 表的直接数据库操作。它提供了完整的增删改查 (CRUD) 功能，包括 addCustomer 用于插入新记录 、getCustomer 和 getCustomerByName 用于按ID或用户名查询 、updateCustomer 用于更新现有记录 以及 deleteCustomer 用于移除指定客户 。该层特别强调了使用 PreparedStatement 来有效防止 SQL 注入攻击，并通过参数化查询提高了操作的安全性与健壮性 。此外，还提供了 getAll 方法来检索所有客户记录 ，以及一个复杂的 getCustomersDetailed 方法，通过多表联接聚合客户的订单、商品、商家和配送状态等详细信息。

业务逻辑层由 CustomersService 接口及其实现类 CustomersServiceImpl 构成。CustomersServiceImpl 封装了更高级的业务操作，并依赖于 CustomersDAO 进行数据交互。为了优化性能，该服务层广泛应用了 Spring Cache 机制，例如：addCustomer 和 deleteCustomer 操作会清除相关缓存 (@CacheEvict) 以确保数据一致性；updateCustomer 操作会更新缓存中的相应条目 (@CachePut)；而 getCustomer 等查询操作则会利用缓存 (@Cacheable) 来减少对数据库的直接访问。此外，CustomersService 还抽象出 getCommoditiesByCustomerId 等方法，提供与客户相关联的商品信息查询能力。

顶层的 CustomerController 是一个 RESTful API 控制器，负责处理客户端的 HTTP 请求。它通过 @RestController 和 @RequestMapping("/api/customer") 注解暴露接口，将业务逻辑层的服务映射到具体的 HTTP 端点。例如，getCustomer 和 getAllCustomers 对应 GET 请求，addCustomer 对应 POST 请求，updateCustomer 对应 PUT 请求，而 deleteCustomer 对应 DELETE 请求。控制器层通过 @RequestParam、@PathVariable 和 @RequestBody 等注解灵活地接收请求参数和请求体，并将处理结果封装在 ResponseEntity 中返回，确保了与客户端高效且标准的交互。整个实验过程通过详细的控制台输出和数据库截图验证了增删改查功能的成功实现，证明了系统设计的有效性和正确性 。