1 Design and Implementation

1.1 Back-end Design

1.1.1 Development framework

Our software is implemented under the *Spring Boot* framework[1] which excels in quick development along with a large mount of community resources and third-party packages.

Since there are geometric requirements on data, we have chosen *PostgreSQL* as our database repository for its built-in geometry functions. To communicate with database, we have chosen *MyBatis* as the Object-Relational Mapping (ORM) framework. Comparing to other choices, such as *Hibernate* and Java Persistence API (JPA), *MyBatis* is more low-level and we can control all SQL statements hence are equipped to fine-tune them to optimise in efficiency, rather than automatic generated SQL interfaces by other frameworks.

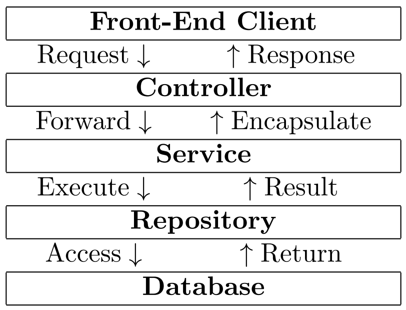
1.1.2 Layered Architecture

Our back-end service is separated into below three layers following common industrial patterns[2, 3, 4]:

- **Presentation Layer (Controller)**: Receive HTTP requests from front-end clients, forward them to the service layer and return the handled results as HTTP responses.
- **Business Layer (Service)**: Handle logic and calculations, and return the weaved results.
- **Persistent Layer (Repository)**: Ask the database to perform specific SQL statements and obtain the results.

Each layer invokes only the next layer, and cross-level invoking is not allowed. A layer focus solely on its duty, clarifying themselves while lowering code coupling. These separation of concerns significantly increases maintainability and reusability of code.

A sketch illustration describing the data flow among the layers is as follows:



1.1.3 *Spring*'s Dependency Injection & Inverse of Control

According to the "Composition over inheritance" ideas, *Spring* has introduced the inverse of control (IoC) realised by dependency injection (DI). Under this framework developers define interfaces as proclamation of service functionalities visible to other modules who only claim dependency on interface level and do not concern actual implementations.

The first merit of introducing IoC is the increase of maintainability since we could minimise the effort on other modules depending on an interface, as long as the interface offers immutable responsibility.

The service and repository layer of our software accordingly applies the IoC principle described above.

1.2 Back-end Implementation: Controller

1.2.1 REST API

The APIs of our back-end service apply to the REST API style. Among the REST principles the most fundamental one states that, prefer HTTP request methods with distinct verbs mapping to one URI instead of embedding verbs into URIs. Four widely used methods with sample URIs are listed and explained as below:

- `GET /user/uid/{id}`: To get details of a user of user ID `id`.
- `POST /user`: To insert a new user.
- `PUT /user/uid/{id}`: To update the user of user ID `id`.
- `DELETE /user/uid/{id}`: To delete the user of user ID `id`.

These APIs with similar URIs are differentiated by methods. In some circumstances, we also use `HEAD` methods to retrieve meta-data. Among them, `GET`, `HEAD`, `PUT` and `DELETE` methods should be idempotent, referring to that "the intended effect on the server of multiple identical requests with that method is the same as the effect for a single such request"[5] as stated by the official RFC 7231 document. In other words, sending idempotent requests multiple times continuously does not change the status of the resource on the server any farther. However, browsers usually do not accept the request body of a `GET` method, thus we also use `POST` methods to query resources with complex conditions as a compromise.

The following sees a sample code snippet truncated from our source code, presenting how the above design pattern is implemented:

Listing 1: Sample REST APIs

```java
@RestController
@RequestMapping("/api")
@CrossOrigin
public class UserController {
    ...
    @GetMapping("/admin/user/uid/{id}")
    public ResponseBody getUserByUserId(@PathVariable String id) {
        ...
    }
    @RequestMapping(value = "/admin/user/uid/{uid}",
                    method = RequestMethod.HEAD)
    public void headUserLastModifiedByUID(
                    HttpServletResponse response,
                    HttpServletRequest request,
                    @UserUID @PathVariable String uid,
                    @RequestParam(value = "time",
                                  required = true) Long timestamp
                                        ) {
        ...
    }
    @PostMapping("/admin/user")
    public ResponseBody insertUsers(@RequestBody List<User> list) {
        ...
    }
    @PutMapping("/admin/user")
    public ResponseBody updateUsers(@RequestBody List<User> list) {
        ...
    }
    @DeleteMapping("/admin/user")
    public ResponseBody deleteUserByUserIds(@RequestBody Map<String
        , Object> body) {
        ...
    }
    ...
}
```

These methods return HTTP responses with JSON bodies stringified from the encapsulated `ResponseBody` objects, which helps front-end service analyse obtained data in a united manner.

1.2.2 Identification

To keep track on the identity of front-end clients among stateless requests and responses, we have made use of the JSON Web Token (JWT). A client needs to first access the path `/api/login` to get a JWT as browser cookie. When the client tries to access any path requiring identification, the `TokenFilter` gets activated and verifies whether the JWT provided as cookie is valid or not. If the client visits resources which are confined to administrators, the `SpInterceptor` gets activated after passing the filter and verifies whether the JWT says the user is an administrator or not.

To simplify code and enhance maintainability we have introduced Aspect-Oriented Programming (AOP). Take the following code snippet as an example:

Listing 2: Sample of implicit identification by AOP

```
    @UserIdentificationExecution
    @AssetOwnershipExecution
    @GetMapping("/user/uid/{uid}/asset/{assetId}")
```

```java
    public ResponseBody getMyAssetByAssetIdWithUID(
                HttpServletResponse response,
                HttpServletRequest request,
                @UserUID @PathVariable String uid,
                @UserAssetId @PathVariable String assetId) {
        return new ResponseBody(Code.SELECT_OK, List.of(
            assetService.getAssetWithWarningsById(assetId)));
    }
```

The content of the method itself only forwards request and returns response and does not execute anything else explicitly. However, to grant the right of accessing an asset we must first check whether the asset is owned by the given user, which is achieved by Java's dynamic proxy introduced by the following AOP aspect:

Listing 3: AOP Implementation

```java
@Component
@Aspect
public class UserIdentificationAdvice {
    ...
    @Before("@annotation(uk.ac.bristol.advice.
        AssetOwnershipUIDExecution)")
    public void userAssetOwnershipByUID(JoinPoint jp) {
        ...
        if (!QueryTool.verifyAssetOwnership(
                (String) parameters.get("assetId"),
                (String) parameters.get("uid"),
                null)) {
            throw new SpExceptions.ForbiddenException("Asset␣owner␣
                identification␣failed");
        }
    }

    ...
}
```

Since this advice class is annotated by `@Aspect`, methods from this class will be executed whenever the cut points are activated, in this case, before when any method annotated by `@AssetOwnershipExecution`. `@Before` declares that the method annotated by it will be executed before the actual method.

1.2.3 Exception Handling

Exceptions (errors in Java dialect) are thrown all the way up to the controller layer and handled by a class with `@RestControllerAdvice` annotation. Apparently it is realised using AOP knowing by reading the name of the annotation.

A typical exception handler method is as follows:

Listing 4: Exception handler

```java
    @ExceptionHandler({SpExceptions.NotFoundException.class})
    public ResponseBody handleNotFoundException(HttpServletResponse
         response, SpExceptions.NotFoundException e) {
```

```
        e.printStackTrace();
        response.setStatus(HttpServletResponse.SC_NOT_FOUND);
        return new ResponseBody(Code.NOT_FOUND, null, e.getMessage
            ());
    }
```

These handlers also return `ResponseBody` objects, aligning with the standard.

Notice that we have defined a `code` field for the `ResponseBody` objects. This is a refined approach beyond normal HTTP status code, with great system extensibility. For example, when a query failed, we could set `code` to be `Code.USER_NOT_FOUND` if the owner of an asset could not be found, or `Code.FIELD_MISSING` if the asset is not owned by any user, while all of two errors might go to the same HTTP status 400 bad request. This offers front-end more robustness against possible query issues.

It is noteworthy that under deployment environment, the back-end service is not supposed to return any error messages in detail. We have achieved this by reading the identifier `active-env` from the configuration files `application-*.yml`.

1.2.4 API Testing Tool

We have utilised ApiFox to provide sample instances of APIs.

1.3 Back-end Implementation: Service

1.3.1 Spring Boot Transaction

It is important to keep the integrity of data when accessing databases and to achieve which a basic tool is the transaction. A transaction is a sequenced set of operations that either succeed or fail altogether. It is best practice to introduce transactions in the service layer since it is the place where database accessing operations are grouped into a meaningful goal.

The basic workflow concerning transactions for a service method, is that first starts a new transaction, rollback if exceptions are thrown, otherwise go over the method and end the transaction. The process is the same for any service method and could be easily wrapped into aspects. Here we use *Spring*'s built-in `@Transactional` annotation which is implemented with AOP.

When a transactional service method calls another, it is vital to understand how transactions are propagated. We briefly introduce four most commonly used *Spring*'s propagation options here:

- `REQUIRED`: appends to the current transaction if exists otherwise opens up a new transaction.
- `REQUIRES_NEW`: suspends the current transaction and opens up a new transaction every time.
- `SUPPORTS`: appends to the current transaction if exists otherwise does not open up any new transaction.
- `NESTED`: opens up new nested transaction when already in a transaction, and will rollback to when the nested transaction began.

To illustrate our approach, we present the following two cases:

Listing 5: Transactional Service Methods

```
@Service
public class AssetServiceImpl implements AssetService {
    ...
    @Transactional(propagation = Propagation.REQUIRED, readOnly =
        true)
    @Override
    public List<Asset> getAssets(Map<String, Object> filters,
                                 List<Map<String, String>>
                                     orderList,
                                 Integer limit,
                                 Integer offset) {
        return assetMapper.selectAssets(
                QueryTool.formatFilters(filters),
                QueryTool.filterOrderList(orderList, "assets"),
                limit, offset);
    }
    ...
    @Transactional(rollbackFor = Exception.class, propagation =
        Propagation.REQUIRED)
    @Override
    public int insertAsset(Asset asset) {
        ...
    }
    @Transactional(rollbackFor = Exception.class, propagation =
        Propagation.REQUIRES_NEW)
    @Override
    public int deleteAssetByIDs(String[] ids) {
        ...
    }
    ...
}
```

This concludes our transactional strategy, that is, no rollback for pure "get" operations with read-only option to improve performance, and rollback for modifying operations. Notice that we keep the propagation option to be `REQUIRED` for read and write cases and `REQUIRES_NEW` for delete cases since this is the most intuitive approach. There are exceptional circumstances that optimises using other propagation options, for example when we need to send an email after updating data in one service method but fails to send the email. This also leads to a rollback for the updating, even if which is correctly performed. However, to reduce the cost of maintaining our software, we still keep the `REQUIRED` option and manually deal with these cases.

We stick to *PostgreSQL*'s default isolation level `read-committed` which balance between data consistency and performance of accessing, and is appropriate since we estimate and expect that read over write operations for our software.

1.3.2 Complex Queries

We have defined entries of query under arbitrary condition by parsing a Java `Map`, which is a set of key-value pairs. As the method `getAssets` we presented from Listing 5, it receives a `filters` object as parameter. The utility function `formatFilters` then format the object into a string after checking whether it aligns with our standard. For controller APIs, they get JSON data from front-end client, while as known JSON data are also key-value pairs so it is natural to convert a chunk of JSON into a Java `Map`. We provide a valid filter JSON below with all supported operations:

Listing 6: Filter JSON

```json
{
    "filters": {
        "asset_status": "active",
        "asset_type_id": {
            "op": "in",
            "list": [
                "type_003",
                "type_001",
                "type_002",
                "false-undefined-type"
            ]
        },
        "asset_installed_at": {
            "op": "range",
            "min": "1990-01-01",
            "max": "2010-01-01"
        },
        "asset_id": {
            "op": "like",
            "val": "asset_0%"
        },
        "asset_location": {
            "op": "notNull"
        },
        "warning_id": {
            "op": "isNull"
        }
    }
}
```

These fields are self-documented and easy to understand.

To prevent SQL injection we strictly validate all filter parameters before constructing SQL statements. The utility function not only checks the format of each filter field but also validates that:

- The field name must be a valid and allowed column.
- The operation type (`op`) must be one of the supported safe operations (e.g., `=`, `in`, `range`, `like`, `notNull`, `isNull`).
- Type of the values (strings, lists, numbers) must conform to expected formats.

To check whether a field is a valid column, we do not predefine a whitelist, but generate one dynamically during runtime using *PostgreSQL*'s built-in meta data `information_schema.columns` since there is a scalable number of tables and columns in the database, and then we manually exclude sensitive columns.

To prevent raw input from being directly interpolated into SQL statements, the query string is assembled to parametrised placeholders rather the final `where` clause after these validations. For example, the condition on the column `asset_type_id` will become

Listing 7: Parametrised SQL Clause String

```
'
    ...
    and asset_type_id in (#{param2},#{param3},#{param4},#{param5})
    ...
'
```

where the `params` will be passed in using a `Map`. This approach, combined with *MyBatis*'s native support for prepared statements, effectively mitigates the risk of SQL injection while maintaining flexibility for complex dynamic queries.

1.3.3 Scheduled Poller, Notification Sender

1.4 Back-end Implementation: Repository & Database

1.4.1 Dynamic SQL

We have mentioned the complex query using filters. It is implemented using *MyBatis*'s dynamic SQL tags:

Listing 8: Dynamic SQL

```
<sql id="filtering">
    <where>
        <if test="filterString␣!=␣null␣and␣filterString␣!=␣''␣">
            ${filterString}
        </if>
    </where>
</sql>

<sql id="orderAndPage">
```

```
    <if test="orderList␣!=␣null␣and␣orderList.size()␣>␣0">
        order by
        <foreach collection="orderList" item="item" separator=",">
            ${item.column} ${item.direction}
        </foreach>
    </if>
    <if test="limit␣!=␣null">
        limit #{limit}
        <if test="offset␣!=␣null">
            offset #{offset}
        </if>
    </if>
</sql>

<select id="selectAssets" resultMap="ResultMaps.AssetMap">
    select asset_id,
           asset_name,
           asset_type_id,
           asset_owner_id,
           ST_AsGeoJSON(asset_location) as asset_location,
           asset_capacity_litres,
           asset_material,
           asset_status,
           asset_installed_at,
           asset_last_inspection,
           asset_last_modified,
           asset_types.*
    from assets
             left join asset_types
                       on asset_type_type_id = asset_type_id
    <include refid="ResultMaps.filtering"/>
    <include refid="ResultMaps.orderAndPage"/>
</select>
```

This design excels in reusability by making filtering and ordering `<sql>` blocks. While obtaining flexibility, security must be assured. The prevention on SQL injection for the filtering block is already discussed before, and it is similar for the ordering block.

There are still security risks exposing the name of columns to the front-end. A safer approach is to predefine a mapping relation between column names and their aliases, for example `user_name -> name` and only the aliases are exposed to the front-end. Though this approach leads to another issue: duplicate aliases for joint table queries. If `asset` and `user` both have IDs, we then have to set their aliases along with table names, which makes aliases little differentiated with original names.

1.4.2 Pagination

Consider the following situation. After polling weather warning data from Met Office site, we want to send emails to users who own assets intersecting with the warning. If there are a large amount of users and we fetch them in one shot, the application will slowdown on performance significantly and might even crash. Hence it is vital to restrain the size of retrieved data from database every time using `limit` to avoid out of memory.

At controller level, we achieve this by going through a barrier: if `limit` is not received as a URI path parameter or request body JSON field, set it to a predefined value, say 1000; if `limit` is received but larger than 1000, then we set it to 1000 and insert a warning message into the response body.

At service level, the story is similar: fetch limited data every time in a loop. A sample code snippet is provided below.

Listing 9: Sample Pagination at Service level

```
private void handleGroupedUsersWithRespectToPagination(Warning
    warning, boolean getDiff) {
    int limit = Code.PAGINATION_MAX_LIMIT;
    long cursor = 0L;
    int length = 0;
    do {
        // get limited data every go
        List<UserWithAssets> list = userService.
            groupUsersWithOwnedAssetsByWarningId(
            limit, cursor, warning.getId(), getDiff, warning.
                getAreaAsJson()
        );
        length = list.size();
        if(length == 0) break;
        // send emails to obtained users
        for (UserWithAssets uwa : list) {
            contactService.sendNotificationsToUser(warning, uwa
                );
        }
        // move the cursor forward
        cursor = userService.getUserRowIdByUserId(list.get(list
            .size() - 1).getUser().getId());
    } while (length > 0);
    ...
}
```

Notice that we used `cursor` realised with user row ID, which is the auto-increasing big integer type primary key for the user table. Instead of using `offset`, cursor prevails on performance since it queries the index on the primary key, while `offset` is achieved by skipping records which is of $O(N)$ time complexity.

It is also necessary to restrict the usage of `offset` at controller level. For deep pages (∼50th page) we do not allow arbitrary page jumping, ”to next page” is the only allowed way of accessing further data, which is implemented using cursor. Nonetheless, jumping to a specific page that satisfies some condition, for example, to the page that users are registered after `'2025-09-01'`, could be allowed once with effective indices since it is equivalent to jumping to the first page controlled by a `where` condition.

1.4.3 Optimisations

As known most modern persistent database frameworks store data using `B+` tree by default. A `B+` tree grows on the primary key of a table. Say that we have a table `users` with primary key `int row_id`. When we want to find a user with a given `row_id`, the time complexity of finding it is exactly the time complexity of this tree, which is $O(\log_f N)$ where f is the fan width of the tree and N is the total size of data. Now, say we want to find users that are above 20 years old. Without indices, the only possible approach is to scan all data which is of time complexity $O(N)$. To optimise, we need to maintain another `B+` tree ordered by users' ages. Exactly this describes how mainly we optimise on database queries, that is, by defining appropriate `B+` trees, which are dubbed indices.

References

[1] VMWare Inc. *Spring Boot Official Online Documentation Homepage.* https://docs.spring.io/spring-boot/index.html

[2] Martin Fowler. *Patterns of Enterprise Application Architecture.* Addison-Wesley, 2002.

[3] Eric Evans. *Domain-Driven Design: Tackling Complexity in the Heart of Software.* Addison-Wesley, 2003.

[4] Mark Richards. *Fundamentals of Software Architecture: An Engineering Approach.* O'Reilly Media, 2020.

[5] R. Fielding, etc, *RCF 7231: Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content.* https://www.rfc-editor.org/rfc/rfc7231