Contents

MoveRest

MoveApiClient

```java
@ApplicationScoped
public class MoveApiClient extends ApiClientBase {
    1 usage  ThomasWiddowson
    @Produces(TEXT_PLAIN)
    public CompletableFuture<String> getMoveData(String moveName) {
        return getData( endPoint: "move/" + moveName);
    }
}
```

Gets all the move data from the PokeAPI for a specific move name in the form of a json string using the 'move' endpoint and this is done asynchronously.

MoveRepository

Handles all the CRUD operations for moves and any other database operations and logic.

MoveService

```java
@ApplicationScoped
public class MoveService {
    1 usage
    @Inject
    MoveParser moveParser;
    1 usage
    @Inject
    MoveApiClient moveApiClient;

    1 usage  ThomasWiddowson
    public CompletableFuture<Move> createMove(String moveName) {
        return moveApiClient.getMoveData(moveName)
                .thenApply(jsonresponse -> moveParser.parse(jsonresponse));
    }
}
```

Handles all the business logic for moves. The move service injects both the API client and the Move Parser. The creation of a move is done asynchronously as not to block the main thread.

Parse

APIParser

```
public interface APIParser<Variable> {
    2 usages  2 implementations  ThomasWiddowson
    Variable parse(String jsonResponse);
}
```

Is the interface that gets implemented by both Move parser and Pokémon parser that takes a string as a parameter and returns a variable.

MoveParser

```
@Override
public Move parse(final String jsonResponse) {
    JsonObject moveJson = Json.createReader(new StringReader(jsonResponse)).readObject();

    String mName = moveJson.getString( s: "name");

    int mAccuracy = 0;
    JsonValue accuracyValue = moveJson.get("accuracy");
    if (accuracyValue instanceof JsonNumber) {
        mAccuracy = ((JsonNumber) accuracyValue).intValue();
    }

    int mPower = 0;
    JsonValue powerValue = moveJson.get("power");
    if (powerValue instanceof JsonNumber) {
        mPower = ((JsonNumber) powerValue).intValue();
    }

    int mPP = 0;
    JsonValue ppValue = moveJson.get("pp");
    if (ppValue instanceof JsonNumber) {
        mPP = ((JsonNumber) ppValue).intValue();
    }

    Types mTypes = Types.NORMAL;
    JsonObject typeObject = moveJson.getJsonObject( s: "type");
    if (typeObject != null) {
        String typeName = typeObject.getString( s: "name",  s1: "normal"); // Default to "normal" if missing
        mTypes = Types.valueOf(typeName.toUpperCase());
    }

    return new Move(mName, mTypes, mPower, mAccuracy, mPP);
}
}
```

Handles all the parsing logic by passing in a string and then grabbing all the details that are needed to create the move object and returns those details as an object. Those details being Name, Accuracy, Power, PP and type.

PokemonParser

```java
@Override
public Pokemon parse(final String jsonResponse) {
    JsonObject pokemonJson = Json.createReader(new StringReader(jsonResponse)).readObject();

    PokemonBuilder pokemonBuilder = pokemonBuilderProvider.get();

    String pName = pokemonJson.getString( s: "name");


    List<Types> pTypes = pokemonJson.getJsonArray( s: "types").stream() Stream<JsonValue>
            .map(typeValue -> {
                String typeName = typeValue.asJsonObject()
                        .getJsonObject( s: "type")
                        .getString( s: "name").toUpperCase();

                return Types.valueOf(typeName);
            }) Stream<Types>
            .toList();
    Stats pStats = new Stats(
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 0).getInt( s: "base_stat"), // hp
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 1).getInt( s: "base_stat"), // attack
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 2).getInt( s: "base_stat"), // defense
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 3).getInt( s: "base_stat"), // special-attack
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 4).getInt( s: "base_stat"), // special-defense
            pokemonJson.getJsonArray( s: "stats").getJsonObject( i: 5).getInt( s: "base_stat")  // speed
    );
```

```java
    Map<String, Integer> pMoveSet = pokemonJson.getJsonArray( s: "moves").stream() Stream<JsonValue>
            .map(moveValue -> {
                JsonObject moveObject = moveValue.asJsonObject().getJsonObject( s: "move");
                String moveName = moveObject.getString( s: "name");

                JsonArray versionGroupDetails = moveValue.asJsonObject().getJsonArray( s: "version_group_details");
                JsonObject latestVersion = versionGroupDetails.getJsonObject( i: versionGroupDetails.size() - 1);
                int levelLearned = latestVersion.getInt( s: "level_learned_at");
                String learnMethod = latestVersion.getJsonObject( s: "move_learn_method")
                        .getString( s: "name");


                if (Objects.equals(learnMethod, ("level-up"))) {
                    return Map.entry(moveName, levelLearned);
                }else {
                    return null;
                }

            }) Stream<Entry<...>>
            .filter(Objects::nonNull)
            .collect(Collectors.toMap(
                    Map.Entry::getKey,
                    Map.Entry::getValue
            ));

    String pFrontSprite = pokemonJson.getJsonObject( s: "sprites").getString( s: "front_default");
    String pBackSprite = pokemonJson.getJsonObject( s: "sprites").getString( s: "back_default");

    return pokemonBuilder
            .setName(pName)
            .setBackSpriteURL(pBackSprite)
            .setFrontSpriteURL(pFrontSprite)
            .setLevel(5)
```

Handles all the parsing logic for the Pokémon objects by taking in a string which is the json response from the Api client and grabbing all the details that are needed for the Pokémon builder to build a Pokémon object. The details being Name, Stats, Types and MoveSet.

PokemonRest

PokemonApiClient

```java
@ApplicationScoped
public class PokemonApiClient extends ApiClientBase {
    1 usage  ThomasWiddowson
    @Produces(TEXT_PLAIN)
    public CompletableFuture<String> getPokemonData(String id) {
        return getData( endPoint: "pokemon/" + id);
    }
}
```

Gets all the Pokemon data from the PokeAPI for a specific Pokemon id in the form of a json string using the ‘pokemon’ endpoint and this is done asynchronously.

PokemonRepository

```java
@ApplicationScoped
public class PokemonRepository {
    1 usage
    static EntityManagerFactory emf = Persistence.createEntityManagerFactory( persistenceUnitName: "default");
    7 usages
    static EntityManager em = emf.createEntityManager();
    1 usage  ThomasWiddowson
    public Pokemon savePokemon(Pokemon pokemon) {
        em.getTransaction().begin();
        em.persist(pokemon);
        em.getTransaction().commit();
        return pokemon;
    }
    no usages  ThomasWiddowson
    public List<Pokemon> findAll() { return em.createQuery( qlString: "SELECT p FROM Pokemon p", Pokemon.class).getResultList(); }
    2 usages  ThomasWiddowson
    public Optional<Pokemon> findById(Long id) {
        return Optional.ofNullable(em.find(Pokemon.class, id));
    }
    no usages  ThomasWiddowson
    public void delete(Long id) {
        var pokemon = findById(id);
        em.remove(pokemon);
    }
    no usages  ThomasWiddowson
    public Pokemon update(Pokemon pokemon) { return em.merge(pokemon); }

}
```

Handles all the CRUD operations for Pokemon and any other database operations and logic using the entity manager.

PokemonService

```java
@ApplicationScoped
public class PokemonService {

    1 usage
    @Inject
    PokemonApiClient pokemonApiClient;
    1 usage
    @Inject
    PokemonParser pokemonParser;
    2 usages
    @Inject
    PokemonRepository pokemonRepository;
    2 usages  ThomasWiddowson
    public CompletableFuture<Pokemon> createPokemon(String id) {
        return pokemonApiClient.getPokemonData(id)
                .thenApplyAsync(jsonResponse -> pokemonParser.parse(jsonResponse));

    }
    1 usage  ThomasWiddowson
    public void savePokemon(Pokemon pokemon) { pokemonRepository.savePokemon(pokemon); }

    1 usage  ThomasWiddowson
    public Pokemon findPokemonById(Long id) {
        return pokemonRepository.findById(id)
                .orElseThrow(() -> new NoSuchElementException("Pokemon with ID " + id + " not found."));
    }
}
```

Handles all the business logic for Pokemon. The Pokémon service injects the API client, the Pokemon Parser and the Pokemon Repository. The creation of a Pokémon is done asynchronously as not to block the main thread.

ApiClientBase

```java
public class ApiClientBase {
    1 usage
    private static final Client client = ClientBuilder.newClient();
    1 usage
    private static final String API_URL = "https://pokeapi.co/api/v2/";

    2 usages  ThomasWiddowson
    @Produces(TEXT_PLAIN)
    public static CompletableFuture<String> getData(String endPoint){
        return CompletableFuture.supplyAsync(() -> {
            WebTarget target = client.target( uri: API_URL + endPoint);
            Response response = target.request().get();
            if (response.getStatus() == 200) {
                return response.readEntity(String.class);
            } else {
                return "Error: " + response.getStatus();
            }
        });
    }
}
```

This is a super class that has a method which handles making requests to the PokeAPI asynchronously by specifying the URL I'm targeting and checking whether the connection was successful or not.

Battle

BattleService

```java
@ApplicationScoped
public class BattleService {
    1 usage
    private Pokemon playerPokemon;
    1 usage
    private Pokemon encounterPokemon;
    1 usage
    @Inject
    BattleUI battleUI;

    no usages  new *
    public void initializeBattle(Pokemon playerPokemon, Pokemon encounterPokemon) {
        this.playerPokemon = playerPokemon;
        this.encounterPokemon = encounterPokemon;
        battleUI.updateBattleUI(playerPokemon, encounterPokemon);
    }
    no usages  new *
    public void speedCheck(Pokemon playerPokemon, Pokemon encounterPokemon) {
        if(playerPokemon.getStats().getSpeed() > encounterPokemon.getStats().getSpeed()) {
        }
    }
}
```

Currently still being developed but will hold all the business logic for the battles being stating the battle and checking stats and making any changes due to damage. This class injects the battleUI to be able to update the Ui when each battle starts.

DamageCalculator

```java
public class DamageCalculator {

    2 usages  ThomasWiddowson
    public static int calculateDamage(Move move, Pokemon opponent, Pokemon defender) {
        double baseDamage = (double) (((2 * opponent.getLevel() + 2) / 5) * move.getPower() * (opponent.getStats().getAttack() / defender.getStats()
                .getDefence())) / 50 + 2;

        double modifier = TypeEffectivenessService.getEffectiveness(move.getType(), defender.getTypes());

        return (int) (baseDamage * modifier);

    }
}
```

Handles all the calculations for how much damage a move will do to a Pokémon by taking the move used as a parameter as well as the defending Pokémon and the attacking Pokémon and using the formula provided by Pokemon to calculate the damage.

TypeEffectivenessService

```
1 usage  ThomasWiddowson
public class TypeEffectivenessService {
    1 usage  ThomasWiddowson
    public static double getEffectiveness(Types mType, List<Types> pTypes) {
        double modifier = 1.0;

        for(Types pType : pTypes) {
            if (pType.isWeakTo(mType)) {
                modifier *= 2.0;
            } else if (pType.resists(mType)) {
                modifier *= 0.5;
            }

        }
        return modifier;
    }
}
```

Handles all the logic for finding out the type effectiveness that is then used in the damage calculator this is done by taking the moves type in as a parameter as well as the types of the defending Pokémon to find the effectiveness modifier.

Builder

Move

```
@Entity
@Table(name = "move")
public class Move {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    3 usages
    @Column(name = "name")
    private String name;
    3 usages
    @Enumerated(EnumType.STRING)
    private Types type;
    3 usages
    @Column(name = "power")
    private int power;
    3 usages
    @Column(name = "accuracy")
    private int accuracy;
    3 usages
    @Column(name = "pp")
    private int pp;
```

Maps out the Move table in the database and is the class that I create the Move objects from however I will eventually move to using a transfer object and entity to keep things separate.

Pokemon

```java
@Entity
@Table(name = "Pokemon")
public class Pokemon implements Damageable {
    @Id
    @GeneratedValue(strategy = GenerationType.IDENTITY)
    private Long id;
    4 usages
    private String name;
    2 usages
    private String backSpriteURL;
    2 usages
    private String frontSpriteURL;
    2 usages
    private int level;
    7 usages
    private int currentHP;
    3 usages
    @Embedded
    private Stats stats;
    3 usages
    @ElementCollection
    @CollectionTable(
            name = "pokemon_move_sets",
            joinColumns = @JoinColumn(name = "pokemon_id")
    )
    @MapKeyColumn(name = "move_name")
    @Column(name = "level_learned")
    private Map<String, Integer> moveSet;
    3 usages
    @ElementCollection(targetClass = Types.class)
    @CollectionTable(
            name = "pokemon_types",
            joinColumns = @JoinColumn(name = "pokemon_id")
    )
```

```
@Enumerated(EnumType.STRING)
private List<Types> types;
2 usages
@ManyToMany(cascade = CascadeType.ALL)
@JoinTable(
        name = "pokemon_current_moves",
        joinColumns = @JoinColumn(name = "pokemon_id"),
        inverseJoinColumns = @JoinColumn(name = "move_id")
)
private Set<Move> currentMoves;
```

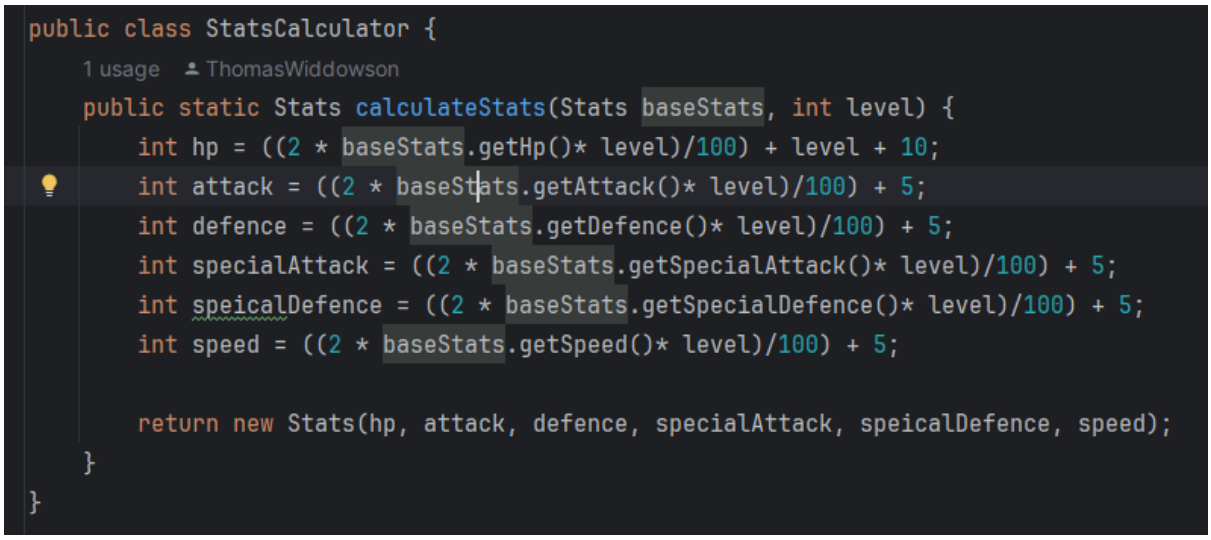
Maps out the Pokemon table in my database and configures all the joining tables and embbeded tables that need to be included. Currently I use this class to make Pokemon objects from however as is the same with the move class I will move to using a transfer object and an entity class to keep things separate.

Stats

```
@Embeddable
public class Stats {
    3 usages
    @Column(name = "special_attack")
    private int specialAttack;
    3 usages
    @Column(name = "special_defence")
    private int specialDefence;
    3 usages
    @Column(name = "attack")
    private int attack;
    3 usages
    @Column(name = "defence")
    private int defence;
    3 usages
    @Column(name = "speed")
    private int speed;
    3 usages
    @Column(name = "hp")
    private int hp;
```

This is the stats class that gets embedded into the Pokemon table, and it holds all the Pokémon's current stats.

Stats Calculator



```
public class StatsCalculator {
    1 usage  ThomasWiddowson
    public static Stats calculateStats(Stats baseStats, int level) {
        int hp = ((2 * baseStats.getHp()* level)/100) + level + 10;
        int attack = ((2 * baseStats.getAttack()* level)/100) + 5;
        int defence = ((2 * baseStats.getDefence()* level)/100) + 5;
        int specialAttack = ((2 * baseStats.getSpecialAttack()* level)/100) + 5;
        int speicalDefence = ((2 * baseStats.getSpecialDefence()* level)/100) + 5;
        int speed = ((2 * baseStats.getSpeed()* level)/100) + 5;

        return new Stats(hp, attack, defence, specialAttack, speicalDefence, speed);
    }
}
```

Handles the logic for calculating a Pokémon's stats based of their level and their base stats which is grabbed from the PokeAPI.

Types

```
26 usages  ThomasWiddowson
public enum Types {
    no usages
    GRASS(new String[] {"Fire", "Ice", "Bug", "Poison", "Flying"}, new String[] {"Water", "Electric"}),
    no usages
    FIRE(new String[] {"Water", "Rock", "Ground"}, new String[] {"Fire", "Bug", "Steel", "Fairy"}),
    no usages
    WATER(new String[] {"Electric", "Grass"}, new String[] {"Fire", "Water", "Ice"}),
    no usages
    ELECTRIC(new String[] {"Ground"}, new String[] {"Electric", "Flying"}),
    no usages
    FLYING(new String[] {"Electric", "Ice", "Rock"}, new String[] {"Fighting", "Bug", "Grass"}),
    1 usage
    NORMAL(new String[] {"Fighting"}, new String[] {"Ghost"}),
    no usages
    BUG(new String[] {"Fire", "Flying", "Rock"}, new String[] {"Fighting", "Grass", "Ground"}),
    no usages
    DARK(new String[] {"Fighting", "Fairy", "Bug"}, new String[] {"Ghost", "Psychic"}),
    no usages
    DRAGON(new String[] {"Ice", "Dragon", "Fairy"}, new String[] {"Dragon"}),
    no usages
    FAIRY(new String[] {"Steel", "Poison"}, new String[] {"Fighting", "Dragon", "Dark"}),
    no usages
    FIGHTING(new String[] {"Flying", "Psychic", "Fairy"}, new String[] {"Normal", "Rock", "Bug", "Steel", "Ice", "Dark"})
    no usages
    GHOST(new String[] {"Ghost", "Dark"}, new String[] {"Normal", "Fighting"}),
    no usages
    ICE(new String[] {"Fire", "Fighting", "Rock", "Steel"}, new String[] {"Grass", "Ground", "Flying", "Dragon"}),
    no usages
    POISON(new String[] {"Ground", "Psychic"}, new String[] {"Fighting", "Bug", "Fairy"}),
    no usages
    PSYCHIC(new String[] {"Bug", "Ghost", "Dark"}, new String[] {"Fighting", "Psychic"}),
    no usages
```

Types is an Enum class that holds all the Pokemon types that are in the Pokemon games as well as their weaknesses and their resistances.

PokemonBuilder

```
11 usages  ThomasWiddowson
@Dependent
public class PokemonBuilder {
    2 usages
    private String name;
    2 usages
    private String backSpriteURL;
    2 usages
    private String frontSpriteURL;
    2 usages
    private List<Types> types;
    2 usages
    private Stats stats;
    2 usages
    private Map<String, Integer> moveSet;
    2 usages
    private Set<Move> currentMoves;
    4 usages
    private int level;

    1 usage
    @Inject
    MoveService moveService;
```

The Pokemon Builder class is a builder design pattern that uses the data grabbed from the PokeAPI to set a Pokémon's details and then at the end builds the Pokémon by calling the Pokémon's constructor in the build method. This class injects the Move service for setting the current moves of the Pokemon as it requires you to create new Move objects.

Encounters

RandomEncounterGenerator

```java
@ApplicationScoped
public class RandomEncounterGenerator {
    1 usage
    @Inject
    PokemonService pokemonService;

    1 usage  ThomasWiddowson
    public Pokemon generateRandPokemon() {
        Random rand = new Random();
        int upperbound = 649;
        Long randomId = (long) rand.nextInt(upperbound);
        return pokemonService.createPokemon(String.valueOf(randomId)).join();
    }
}
```

This class handles the logic for generating a random Pokémon for each battle encounter by generating a random integer and then using the injected Pokemon service to create the Pokemon of that random id.

StarterGenerator

```java
    2 usages
    @Inject
    PokemonService pokemonService;
    1 usage
    @Inject
    BattleUI battleUI;
    1 usage  ThomasWiddowson
    public List<Pokemon> generateStarters() {
        Random rand = new Random();
        int upperbound = 649;
        List<CompletableFuture<Pokemon>> starterList = new ArrayList<>();

        for(int i = 0; i < 3; i++) {
            Long randomId = (long) rand.nextInt(upperbound);
            starterList.add(pokemonService.createPokemon(String.valueOf(randomId)));
        }

        return starterList.stream() Stream<CompletableFuture<...>>
                .map(CompletableFuture::join) Stream<Pokemon>
                .toList();
    }
```

Handles the logic for generating the three random starters at the start of the game and this is done asynchronously so that the API calls can be made concurrently on different threads rather than doing it one after each other making the whole process faster and more efficient.