Repository Layer Tech Stack Report

What they do:

Postgres is an open-source relational database management system, so is needed as part of the project to store tournament, player and team information in tables on the database.
JPA is used to map Java object to tables in a relational database, used because it is less error prone and time consuming that writing SQL queries. JPA allows using databases without using database semantics. JPA is a part of Spring Data.
Spring Data is used to provide a consistent Spring-based programming model for data access while retaining the features of the underlying data store.
The repository is an abstraction of the data layer. It only deals with data access.

How they work:

- Postgres uses a client server model
  - The server process manages the database files by accepting connections from the client and doing actions on their behalf.
  - Can handle multiple concurrent connections
- Spring Data analyses new Repositories
  - Looks at all the methods defined
  - Automatically generates queries from the method names

Spring Data Features:

- Repository and custom object mapping abstractions
  - Spring Data Repositories are and abstraction which reduce the amount of code required for data access layers.
- Uses the repository names to dynamically derive the queries.
- Domain base classes which provide basic properties
  - Domain class is the model and represents a persistent entity that is mapped onto a database.
- Transparent auditing
  - Spring data allows keeping track of the time an entity was created or changed and who did it.

Using JPA:

Creating an entity and it's corresponding repository

```
@Entity
class Team {
        @Id @GeneratedValue(strategy = GenerationType.AUTO)
        private Long id;
        private String name;

        // getters and setters for each attribute also required
}

interface TeamRepository extends Repository<Team, Long> {
        Team save(Team team);
        Optional<Team> findById(long id);
}
```

Defining Repositories:

A query follows the structure - Type<Entity> query(search parameters)

```
interface TeamRepository extends Repository<Team, Long> {
    //Type<Repository> query(search parameters)
        List<Team> findByIdAndName(Long id, String name);
```

```
        // Enables the distinct flag for the query
        List<Team> findDistinctTeamsByIdOrName(Long id, String name);
        List<Team> findTeamsDistinctByIdOrName(Long id, String name);

        // Enabling ignoring case (upper/lower case) for an individual property
        List<Team> findByNameIgnoreCase(String name);
        // Enabling ignoring case for all suitable properties
        List<Team> findByIdAndNameAllIgnoreCase(Long id, String name);

        // Enabling static ORDER BY for a query
        List<Team> findByNameOrderByFirstnameAsc(String name);
        List<Team> findByNameOrderByFirstnameDesc(String name);
}
```

Subject Keywords

`find...by` - typically returns repository type, collection, streamable, or a result wrapper. Can be used in combination with additional keywords
`delete...By` - acts as Delete query either returns void or delete count
`...Disitinct...` - used to return only unique results, should be used between the `find` and `by` keywords

Predicate Keywords

Logical Keyword -> Keyword expression
`AND` -> `And`, `OR` -> `Or`, `AFTER` -> `After`, `CONTAINING` -> `Containing`, `BETWEEN` -> `Between` etc... (there are also alternatives but they all have a version that follows this pattern)

Modifier Keywords

`IgnoreCase` - used for case-insensitive comparison
`OrderBy...` - specify order

Projections

Interface-based Projections

In this example a projection called WinLosses made using an interface makes it so that when retrieving from the entity if used only returns the team name, win count and loss count for the result.

```
interface WinLosses {
        String getTeamName();
        int getWinCount();
        int getLossCount();
}

interface TeamRepository extends Repository<Team, Long> {
        Collection<WinLosses> findByTeamName(String teamName);
}
```

Class-based Projections (DTOs / Data Transfer Objects)

DTOs hold properties for the fields that are going to be retrieved. Used in the same way as interface projections, but without proxying.

DTO projection example:

```
record WinLosses(String teamName, int winCount, int lossCount) {
}
```