Repository Pattern

Updating the Intranet to Use The Repository Pattern

Arthur Clarkson

**4th September 2024

One recurring issue within one of our modules was standardization of data access methods from the business layer. On the intranet, for each database entity, we should have the following methods: List (get a list of records with given filters applied), GetById (get singular record with id), Create (create and return singular record), Update (update and return record), and ArchiveById (give archivedon and archivedby fields a value and return the record). After doing some research the best approach for this was for us to use the repository pattern.

The repository pattern provides an abstraction of data persistence, so that your application can work with a simple abstraction (that your domain model owns) that has an interface approximating that of a collection. Adding, removing, updating, and selecting items from this collection is done through a series of straightforward methods, without the need to deal with database concerns like connections, commands, cursors, or readers.

To persist this throughout our business layers, I decided to create a generic repository interface, of which each business layer class would inherit from, causing errors during compile time if the interface is not implemented:

```
public interface IRepository<TDto, TFilterObject>
{
        public List<TDto> List(TFilterObject filters);

        public TDto GetById(int id);

        public TDto Create(TDto obj);

        public TDto Update(TDto obj);

        public TDto ArchiveById(int id);
}
```

After applying this to our CalendarController business layer class, because it doesn't implement anything on the interface, it throws up errors, meaning that developers have to follow the interface or else their code won't work: