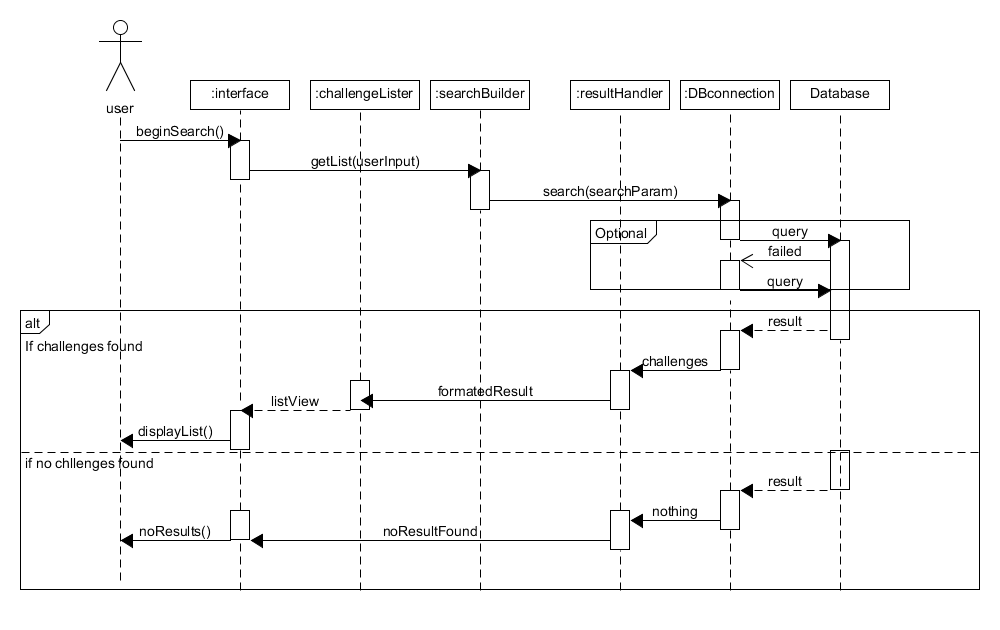
Interaction Diagrams

We focused on a main set of four use cases that represent the core of the program search, view, fetch, and store. This encompasses the basic functions of both our programs front and back ends. It is important that this core is well designed and understood before any future features can be added. The focus of this report will be on a “Guest” user since a registered user is expanded upon a guest user experience we can’t implement those features without this base.

**Use Case 1: Search**

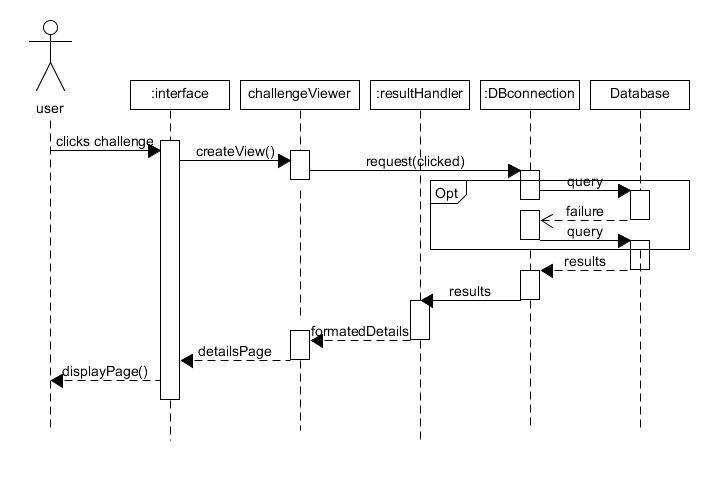
The first and primary use of the application is searching the site. This will be used by both “guest” users and registered users. This begins at the interface which then sends the users requests through all the necessary subsystems. This is similar to Use Case two but has slightly more interaction since it can return a blank result.



**High cohesion principle**: The user requests a search which is passed on to the search builder which represents the high cohesion principle. The interface only handles the interaction between the user and sends requests to other objects to fulfil the request. Also challenge lister is only responsible for creating a list page, if no results are found there is no reason for it to be called. Additionally, separating the result and the challengeLister and resultHandler a change in the format in the database when not affect the creation of a list. Creating high cohesion.

**Expert Doer principle:** By having the DBconnection be the only person that interacts with the database it allows for the other subsystems to not car about how to access the actual database. DBconnection is the only part that accesses in both the front and back ends.

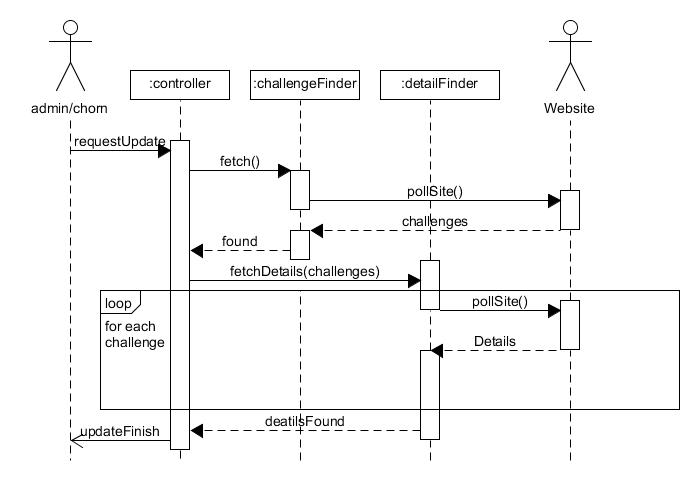
**Use Case 2: View**

 The second most important thing is for users to be able to access the details of the challenges. The interface yet again sends the request through various subsystems. It is important to note that in both use cases that the interface doesn’t manipulate or handle the data directly, but simply interacts with the user presenting the elements created by the system. Since we know that a challenge is stored with details, we know there will not be a case of a blank query, unless there is database malfunction. This is the primary difference between use case one and 2.

**High Cohesion Principle:** When a user clicks on the challenge the interface does request to build the challenge view. By separating the creation and the displaying the application becomes more cohesive.

**Expert Doer Principle:** The results handler handles results from the database no matter what it is since it already knows how to handle challenge input there is no reason to creating another subsystem to handle data from the database.

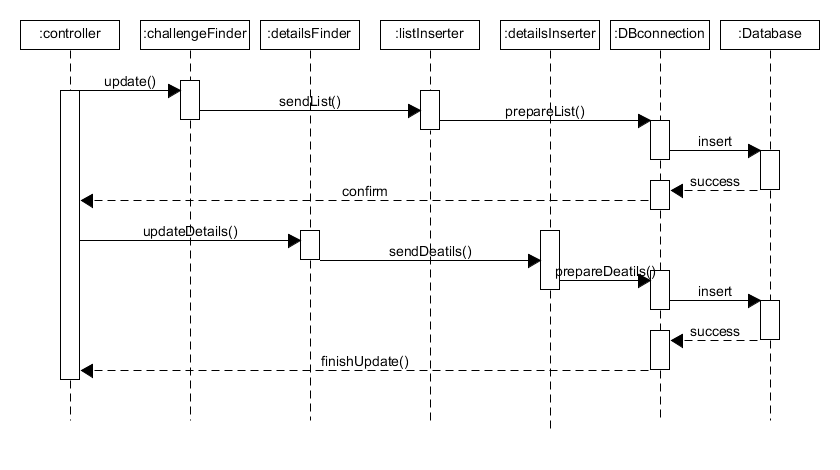
**Use case 7: Fetch**

 Use case 7 and 8 use to be a single use case, but they are both independent enough to be separated. This begins with a request from the system or an admin and then the controller takes over. The controller requests both the challengeFinder and the DetailFinder to scrape the data from the specified website.

**High Cohesion Principle**: Separating the challenge finder and detail finderallows for formatting for two different types of data the location of challenges and the details of the challenge they can be manipulated as needed since they are two very different pieces of data. The controller doesn’t actually manipulate data, but rather control timing and pass messages along to other subsystems.

**Use Case 8: Store**

This case will be automatically called by the system when there is information in finder classes to store. It will then began formatting the data in the inserter objects and the pass it to DB connection to store within the database.

**High Cohesion Principle:** By separating the Challenge locations and the details of those challenges we can format them appropriately without having one big formatting class. This is shown in the separation of detailsInserter and detailsInserter.

**Expert Doer Principle**: Again here we see that DBconnection is the one source that interacts with the database. This makes easy to change interactions with the database since it is all contained within one subsystem.