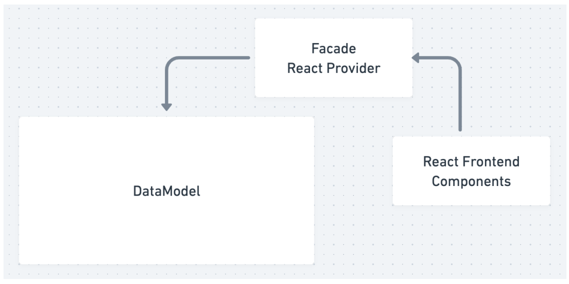
Running The Application

Please read the README.md in the code repository to start the application locally.

Overall Implementation Design



Team-34 decided to design the system in two parts, the DataModel and the Frontend. All of the codebase is written in TypeScript. The frontend uses the React library to render components whilst the DataModel is written in vanilla TypeScript, which follows the OOP principles. The DataModel acts as a backend system which deals with the logic and API calls. The React Provider acts as a Facade between the DataModel and the components such that the components can access the data from the DataModel to display them on the page without directly interacting with the details within the DataModel. The facade is explained in more detail later in this document.

In the system, there are two primary clients: the client package which is a client of the API and the Frontend, which is a client of the facade and thus the DataModel. The client package throughout the documentation and codebase refers to the client of the API within the DataModel.

SOLID Principles

Single Responsibility Principle (SRP)

The SRP states that a class should have one and only one reason to change.

One example of the SRP is the location package, where specific TestSite data and functionality are separated from the Address functionality by having the TestSite extend the Address class. Another example of the SRP is the client package, where functionality for interacting with different parts of the API are separated into different clients; interacting with users, bookings, test sites, tests and more are all separated. Separating the classes in these ways keeps their data hidden from each other to reduce unnecessary exposure to each other and associated classes.

Further classes have been created to encapsulate single functionalities include *BookingCollection* which handles the booking aspect, *UserCollection* which controls the collection of users, *RecordCollection* which handles the state of tests and *TestSiteCollection* which handles the control and manipulation of test sites (e.g. finding nearest sites, gathering their locations and relevant information to provide to user, etc.). These separations avoid the oncoming problem of "god" classes where collections would be doing tasks that are outside of its responsibility (eg. *BookingCollection* displaying nearby test sites as well). This design consideration reduces the complexity of these collection classes and satisfies the requirements; should a user or a facility staff member want to make a booking for example, they would simply interact with the *BookingCollection* to fulfil their tasks. In this way, all collection classes follow the SRP. Thus, changes would not affect other methods or cause ripples of changes to occur if they had existed as one class (code coupling is reduced).

A *Test* class has been created to differentiate a booking from a test, as creating a booking doesn't necessarily equate to a test being undertaken (for instance, in the event that a booked user fails to show up for their COVID test). Furthermore, they are separate sets of information, and storing test data and/or methods would be out of the scope of the *Booking* class. Thus *Test* exists as a class to encapsulate its own data and potential methods in compliance with SRP. Similarly, a *Result* class is included to separate this type of information from the Test class and any relevant methods that should exist e.g. result updates or test updates. The disadvantage of having different classes with single responsibilities which also rely on each other is that the implementation will require a lot more overhead in writing methods to access data, whereas if classes were combined, the methods could directly access the data required. However, the benefit of having reduced coupling allowing for easier feature enhancement far outweighs the cost of a little more overhead.

Liskov Substitution Principle (LSP)

The LSP states that subclasses should remain compatible with the behaviour of their superclasses. Preconditions and parameters cannot be strengthened while postconditions and return types cannot be weakened.

The *Verifiable* abstract class and its subclasses, *QR* and *PIN*, adhere to the LSP. This is required because the *Booking* class needs to associate with both the generic *Verifiable* to store both *QR* and *PIN*. The *QR* and *PIN* must not break any of the LSP checks because the *Booking* class calls generic methods from the *Verifiable* such as “isValid()” to ensure it is valid at the test site and “string()” to provide a string display of the verification. The subclasses have the same parameters and return types as the *Verifiable* super class, thus not breaking any LSP checks.

The *Test* abstract class and its subclasses, *RATTest* and *PCRTest*, adhere to the LSP. Similarly to the *Verifiable*, this is important such that the *RecordCollection* can store both child classes in a single data structure. The RATTest and PCRTest both override “getWaitTime()”, which is a method to get the average waiting time for the result of a test. This method does not have any parameters and both overrides return the same type as the super (number). Thus, this design choice will not break any of the LSP checks.

The *Address* is a class where the details of a location are stored, such as the coordinates and address. Since the *TestSite* stores this information as well and is also a type of *Address*, it simply extends the *Address* class. The LSP checks are met as the *TestSite* does not override anything of the *Address*.

Similarly, the *Person* class meets the LSP as it does not change the parameters or return types of the defined methods in its interface, the *User*.

It is important that the dependents of the factory methods and clients get the correct types because these are what map the information from the API to classes within the DataModel. All factory methods and clients follow the LSP. The interfaces define method parameters and return types and implementations do not alter them; they do often return more specific types, such as the *Person*, *PCRTest, RATTest* and others, but they are all subclasses of the interface defined return type, allowing them to meet the LSP.

Interface Segregation Principle (ISP)

The ISP states that clients should not be forced to depend on interfaces that they do not use and thus abstractions should be segregated to adhere to the SRP.

The client package has been designed such that functionality to retrieve and send data to/from the API are separated based on the information they handle; the booking, user, test site and test data are all handled separately. The authentication is also separated from the *UserClient* because the *UserCollection* does not deal with the login, current user and authentication functionality. These separations are chosen to ensure that collections which retrieve data from the API only needed to know of functionality to handle the data they are interested in (eg. *BookingCollection* only handles booking API data). This is a huge advantage as it ensures the SRP is met, ensuring that changes made to the data handled would not affect other client interfaces. The disadvantage of the separation is that there are many clients to maintain. However, this is much easier than maintaining a huge interface, making the tradeoff well worth it.

The *Authentication* and *DataModel* interfaces are both interfaces which the frontend interacts with to access functionality. They were separated because not all components will require use of both data and authentication; only a few parts of the frontend require the authentication functionality. The *DataModel* contains the four collections. These were not separated because the benefit of the interface segregation is not worth the huge overhead required in creating individual contexts for each of the four collections. Thus, their functionality is kept together.

Dependency Inversion Principle (DIP) and Open/Closed Principle (OCP)

The DIP states that high level modules should not depend on low level modules and instead, both modules should depend on abstractions. Furthermore, abstractions should not depend on details, details should depend on abstractions. The OCP states that software entities (classes, modules, functions, etc.) should be open for extension, but closed for modification.

Instead of having the *RecordCollection* contain *RATTest* and *PCRTest* instances specifically, the *RecordCollection* is a composite aggregation of *Test* instances, a higher level of abstraction. This is possible because the *Test* abstract class contains all of the methods which its dependencies require access to, while the specific *RATTest* and *PCRTest* implementations can override the methods and attributes to more specifically fit their purposes. This adheres to the OCP where the abstraction should be closed for modification, while open for extension of more specific subclasses. For example, the *Test* has the functionality to return the time taken to get the test result, and the *RATTest* overrides this method to return 30 minutes while the *PCRTest* overrides the method to return 3 days. Furthermore, the DIP is adhered to as the *RecordCollection* contains the abstraction while the *RATTest* and *PCRTest* extends it.

A disadvantage of using the DIP here is that if one of the child classes requires additional functionality specific to that test type in the future, the *Test* abstraction and other child classes would need to be updated to support this additional requirement. This is so that the *RecordCollection* can use the additional functionality of the special child class, even though other child classes do not have that capability themselves. The other option is for the special child class to implement another interface, but would mean that the *RecordCollection* would have to implement this interface in some way as well. Both of these options would increase complexity of the system, which could cause issues in the future.

The *User* interface defines general functionality for any user of the system. The *Person* class implements this and contains information specific to a person. It is possible to remove the *User* interface and have the *Person* class on its own. However, this would reduce the extensibility of the system; there may be future requirements for a government agency to access test records, users of which may not want to store personal information in the *Person*. Thus, the OCP is followed such that the system is open to extension through the *User* and closed for modification as it is difficult to change that interface. The DIP is further followed as dependents rely on the *User*, which has specific implementation through the *Person*. It was decided that the specific functionality of the users was not to be done through creating subclasses of the *Person*, such as *HealthcareWorker* and *Patient*. Instead, the types of the user are able to be checked, allowing other dependents to check whether a functionality can be done for a user. This is advantageous since it reduces the dependencies of the user package on others, strengthening stability.

The API client package reflects the DIP and OCP strongly. The DIP is followed as the client interfaces are depended on by the external collection classes and are implemented by *Axios* classes. Furthermore, the OCP is followed as the clients are easily extensible; if different future implementations are required (such as a different API being used), they can simply implement the client interfaces, with the *ClientManager* managing which implementation is used. The main disadvantage is the extra code requirement, but is outweighed by the modularity and extensibility benefits.

The use of the *Verifiable* abstract class implements the DIP and OCP. Here, the *Booking* does not depend directly on the *PIN* and *QR* child classes and instead depends on the *Verifiable*. The Booking stores *Verifiable* instances relevant to itself while the *VerifiableFactory* creates *Verifiable* instances. This design is possible because the *Verifiable* contains the methods and attributes common to all *Verifiable* subclasses, where the *PIN* and *QR* simply override methods to provide more specific functionality, adhering to the OCP. Thus, the DIP is met as the *Patient* and *Booking* classes rely on the high level *Verifiable* which is extended by the more specific *PIN* and *QR* classes.

Package Cohesion Principles

The Reuse/Release Equivalence Principle (REP) states that a unit of reuse (component) cannot be larger than a unit of release. The Common Closure Principle (CCP) states that when a change is required, all classes in a component should be affected together. Lastly, the Common Reuse Principle (CRP) states that classes in a component should be reused together. The version control repository and descriptive documentation allows for packages to be able to trust each other. Developers will be able to read through the documentation to understand how the units in each package are to be used.

The client API classes have good cohesion within their package. There is high level abstraction in the *Client* interface (which defines a base client), further abstractions in the client interfaces for each section of the API and an abstraction to define the functionality of a client manager (CRP). These are all implemented by concrete classes, prefixed with *Axios* (CCP) as Axios is the library used to make HTTP requests.

Factory methods are further segregated into subpackages within the packages of the classes they instantiate. This ensures that only functionality to do with creating a specific class is held within that package. These factory methods have good cohesion through having generics in the *Creator* interfaces (eg. *TestCreator*) and specifics in the concrete classes which implement them (eg. *PCRTestCreator* and *RATTestCreator*).

The *Verifiable* and its implementing classes, *PIN* and *QR*, are reused at two stages in the booking process for the verification of verifiable codes. The *VerificationFactory* is responsible for creation, and the *Verifiable* (PIN and/or QR) have self contained functionality for verification. As all the classes are necessary to perform the verification steps, they have been packaged together within the verification package. This follows CRP in which all classes within the component (package) are reused together if even one is required. Furthermore, there is a good balance between the CRP and CCP in that there is an abstraction which houses common behaviour of the verification process in the *Verifiable* abstract class (CRP), while more specific behaviour is housed in the *PIN* and *QR* classes (CCP).

The user package contains the *User* interface and the *Person* class. These classes are all included in this package because these are the classes which the users interact with to access the functionality of the system. This package maintains cohesion balance in the same way as the verification package; the common behaviour for users is defined in the *User* interface while the package has specific functionality with the *Person* class.

The test, booking and location packages all have well balanced cohesion. The components are grouped based on their functionality: test, booking and location (CRP). The main functionality is dealt with in the collection classes and a generic is present in the *Test* class and is specific through the *Booking*, *TestSite* classes and the *Test* subclasses (CCP).

Package Coupling Principles

The Acyclic Dependencies Principle (ADP) states that the dependency graph of packages and components should not be cyclical. Furthermore, the Stable Dependencies Principle (SDP), packages should only depend on packages more stable than itself, where the stability of a package is a measurement of package change likeliness. Lastly, the Stable Abstractions Principle (SAP) states that the more stable a package is, the more abstract it should be; highly stable units in a design should be made abstractions.

	client	user	location	verification	booking	test	DataModel
Stability	0.72	0.38	0.25	0	0.44	0.43	0.5
Abstractness	0.5	0.33	0	0.29	0	0.22	0.34

The metrics show that the client package is unstable due to many dependencies one external entities. In this case, this is not a bad thing because the external dependencies are primarily on factories and the classes which it returns. Removing these from the picture and the client package is a stable and reliable package. This is important since all collections depend on the clients to interact with the API. The user, location, verification, booking and test packages are all quite stable classes, with most dependencies being to simply store instances of other classes within themselves. All complex functionality are within packages, making each package highly stable. This is important since the packages have a primary access point in the collections, as they are exposed to the frontend through the contexts (facade). This is reinforced by the stability metrics of the location, verification, test and interface packages, which are all 0.5 or lower, meaning that they are mostly independent.

The overall design meets the ADP since there are no dependency circles; all dependencies are primarily within packages, such as collections depending on other internal classes.

Although these packages are highly stable, their abstract metric is low. This is because the verification and test packages contain abstractions which are then extended by various subclasses, but are predominantly hidden from public use. Furthermore, the existence of factories decreases the abstractness due to the various creators required. The booking and location classes do not have any abstractions since their classes require initialisation (the *Address* class may require initialisation to store the user address in the future).

Design Patterns

Factory & Factory Method

The factory method is an interface for creating objects in a superclass where concrete creators define the creation of the concrete classes. The factory is a creational design pattern which depends on creator implementations to instantiate the various concrete classes. These principles are implemented to assist in creating instances of the *User*, *Verifiable* and *Test* abstractions; the API client implementations use the factories to create *Person* (*User*), *PIN* and *QR* (for *Verifiable*) instances and *PCRTest* and *RATTest* instances (for *Test*) depending on what types are provided for each instance of data from the API. This is advantageous because the SRP, DIP and OCP are followed; product creation is handled by separate entities than the clients themselves (SRP), although dependents rely on the clients (DIP), and when new subclasses are created, the creator interfaces can easily be extended to accommodate the new functionality (OCP). The main issue with this design pattern is that various new classes and sub-packages were required to implement the pattern which increased the complexity. Overall, the design choice is beneficial since the readability of the codebase is better and the system is more extensible.

Singleton

The singleton is a creational design pattern which ensures that a class only has a single instance at any time while providing global access to the resource. The singleton is followed in the *AxiosClientManager* (which serves to allow access to the API functionality through various API clients) and all of the factories. The singularity benefit where the class only has one instance at a time and the global accessibility benefit were the driving factors for this design choice. The primary disadvantage is that there are now two reasons for change: lifecycle management and actual functionality. The disadvantage in this case is small and thus the pros outweigh the cons, making the design choice suitable for the system.

Facade

The facade is a structural design pattern which provides a simplified interface to a set of complex classes. The *DataModel* is an interface to the collection classes of the DataModel. Similarly, the *Authentication* interface defines functionality for authentication (logging in, registration, current user). The components in the frontend are able to use these functionality through the *DataModelContext* and the *AuthenticationContext*, both of which are provided through respective providers. Thus, the facade is implemented through the *DataModel* and *Authentication* interfaces. This design pattern was chosen because the facade isolates the code in the DataModel package from the components, providing only the limited functionality which the components require. A disadvantage is the possibility of a god object. However, this is not the case in this design since the facade only provides reference to other functionality within the backend and does not provide extra functionality other than that. Therefore, the facade is a beneficial design choice since it simplifies the dependency between the frontend and the backend.