Software Engineering Project Report

King's College London

Major Project - Parental Education

Client | KCH Paediatric Liver Service
Dr Marianne Samyn

Team |

Jeanne BEYAZIAN	(k1888763)
Ziyang CHEN	(k1894363)
Andrea CIUFFO	(k1893969)
Marek GRZESIUK	(k1889281)
Mohammad ISLAM	(k1891736)
Callum WATKINS	(k1890584)
Kerim YALCHIN	(k1889476)
Yicheng ZHAN	(k1895372)

Product | https://paediatric-liver-service.netlify.com/

Administrator account (for our deployment)

Name: Backend
Email: backend@test.com
Password: backend

Contents

Project Brief

The team has been formed to resolve a KCH Liver Service's issue regarding parent's education on a variety of liver conditions affecting children, particularly Biliary Atresia.

The King's College Hospital Paediatric Liver Service wants to improve the **quality** of **their information delivery**. They wished to **simplify the process** of informing parents about their child's condition as it can be very tedious and exhausting for both nurses and parents. The patient's parents often struggle to absorb all of the information during lengthy meetings with medical staff not long after the stress of surgical operations. Parents should also be able to **find answers to any of their queries** that may arise later on. The written documentation given to them is rich in information and can reveal itself to be very complex. The Paediatric Liver Service at KCH covers numerous other liver diseases in children and thus needs to make sure all patients have access to any advice and information they need.

Therefore, the main objective is to make content **easily accessible**, **readable** and **interactive**. It needs to contain all the necessary information about liver conditions the health staff wish to make available in one single place. This is meant to **facilitate** their work and **clarify** the information for patients. The content must also be **easy to update** by staff. The product could eventually be used in other departments of KCH concurrently.

Outcome

The project was successful as all requirements set by the client were satisfied. In this section, we go in depth about what was accomplished.

The team has produced a **secure web application** allowing nurses to make education on Biliary Atresia and other conditions **available** at all times. The app is addressed to parents and managed by the medical staff at the KCH Paediatric Liver Service. As mentioned earlier, the digital version focuses on interactivity: it is visually engaging, graphic, and readable.

Successfully Implemented features

1. The application displays all **the essential information about any children's liver** condition nurses wish to make available online.
   This data is displayed similarly for every viewer, and does not require logging in.
2. The application contains a **bulletin board on the home page** displaying news articles written by nurses.
3. The nurses have the possibility to **log in to their administrator account**. They gain access to the **editor interface, which permits modifications** to the display of all information.
4. Information is presented to users in a **well organised fashion involving separate sites and pages** for different diseases.
5. The application has access to a **bank of medicines** maintained by medical staff.
6. For each condition, the users are able to **create custom drug charts that can be then downloaded or printed**.

Client discussions

As suggested by the client, we agreed not to make the experience personalised in any way. With about 1,500 patients with varying conditions, it would be impossible to represent each case individually.

In the same way, creating a forum would cause too many issues surrounding misinformation. For safety, it is preferable to simply let staff information be available on the site, along with other resources they recommend, such that patients can contact and read from reliable sources only.

Design

This section describes the design phase of the project. The components of the web application that were implemented are illustrated on UML diagrams for both back-end and front-end, as well as the overall structure of the application.

The general high level design of our application is inspired by the **3-Tier Application architecture** which consists of **presentation tier** frontend work invoking selection of **REST API endpoints**. These connect down to classes in the **logic tier** which query the database or disk storage at the **data tier**.
We used the frameworks **'Hibernate'** and **'Micronaut'** for the back-end development of this application. Hibernate was responsible for mapping entity classes to database tables and allowing us to manage stored data. Micronaut provided us with the means to create REST API endpoints as shown in *Figure (1)*.

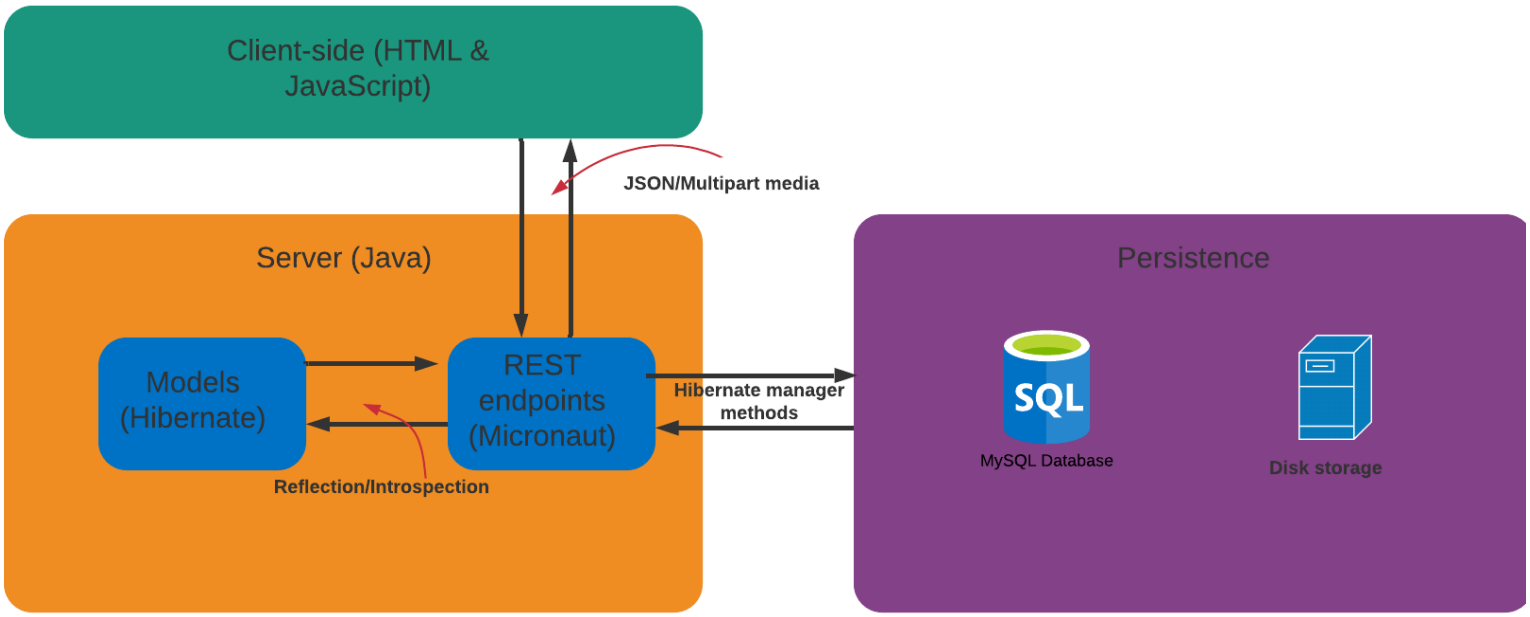

Figure 1 : *Block diagram of the application architecture*

With this architecture, the client may only interact with the server through the REST endpoints implemented in the controller for a specific service. Controllers specify the HTTP requests and the message body format (in our case varying between JSON and Multipart/form-data) as well as the requirement for an X-API Header. The block diagram depicts communications between the server endpoints and persistence via 'manager methods'. These are the business logic classes which carry out all of the transactional work involved in requests. However, Image and AppInfo services have managers that do not communicate with the database as their data is stored server-side. They make use of directory holding classes giving static locations of relevant files and directories. This way, these locations can be changed for testing.

| Back-end Design

The endpoint-manager relationship is shown in *Figure (2).* As our application involves many similar services they can all be roughly represented by a similar template layout. Each class highlighted in red can be thought of as any of our Java service packages.
A single entity, represented by 'ExampleEntity', consists of only basic requests (add, delete and update) for demonstration purposes. Some entities and their managers are more advanced. Through the diagram we can see that all entity controllers share an instance of their associated entity manager interface, which allows the controller to invoke the necessary methods to update the database. This separation of concerns keeps our application scalable and makes it so changes in how the server interacts with the database won't affect how the client interacts with the server.

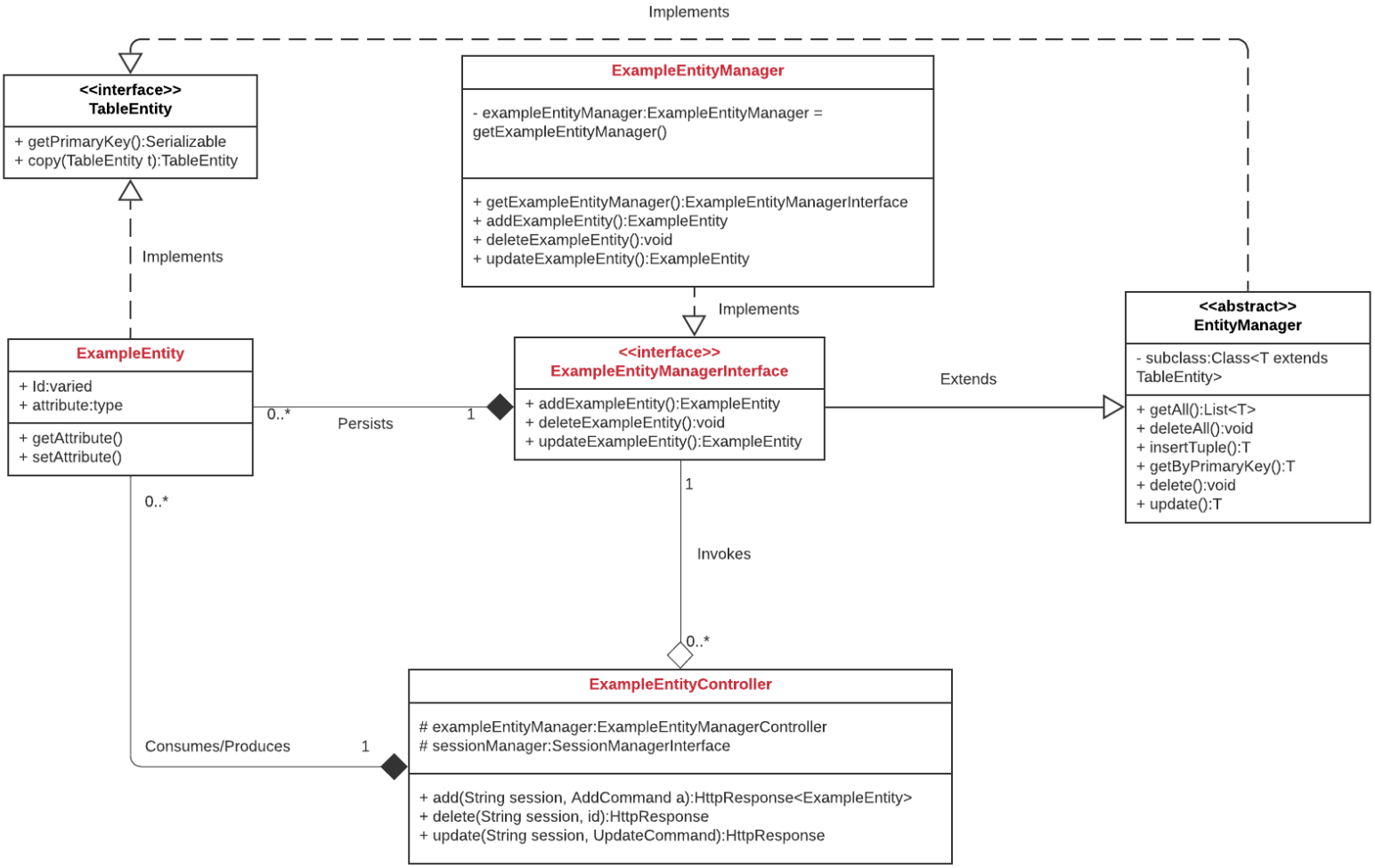

Figure 2 : *Class diagram representing the interaction between entity managers and entity controllers*

In our project we used the **singleton** design pattern and took inspiration from the **command** design pattern. The former is shown in *Figure (2)* as access to the ExampleEntityManager is limited to the 'getExampleEntityManager()' method call, which will ensure that only one instance of the manager is used across the application. The latter is shown in *Figure (3)*, in which we can see that the controller's add and update requests require POJO classes to be passed in.

In encapsulating the commands into these classes, we are improving the application's scalability:

additions to the Entity's fields won't require changes in the controller methods, which would have been the case had we been using the ExampleEntity class to satisfy these requests instead. Update commands simply contain more information (the id) than Add Commands and so one inherits the other - either can be sent into the endpoint selected at runtime (strategy design pattern). POJO basic objects also reduce memory load and easier JSON parse conversion to real objects. Efforts have been made to reduce code duplication and follow clean practises - classes are responsible for their own jobs.

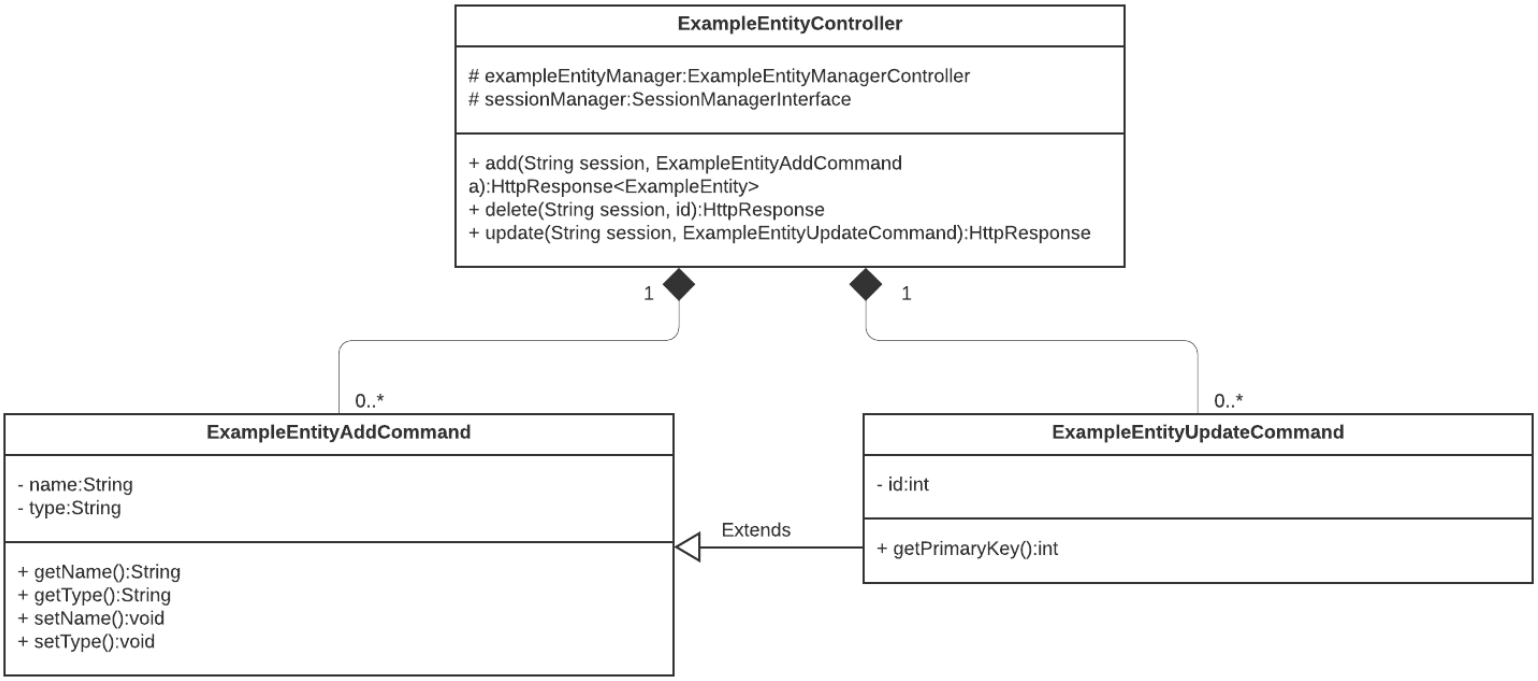

Figure 3 : *Class diagram showing the architecture and role of command classes in our application*

| Front-end Design

The client application was designed to have a simple UI component structure with dedicated services for interacting with the REST API provided by the server application. Vue.js was our modern JavaScript framework of choice for its small size, shallow learning curve, and ability to facilitate rapid development.

Each Vue component encapsulates the DOM structure, styling, and JavaScript logic relevant to the component only. This method of structuring the application into "single file components" has many benefits, including reusability, modularized CSS, and reduced amount of code that must be read to understand each component. Further, the chance of regression is massively reduced when the code is separated in this fashion.

Local browser storage is used where necessary to store data such as API authentication keys and user preferences to ensure that they persist throughout and potentially beyond the user's local session.
Discussing each component or service in detail would be excessive. The following diagram however provides a high-level structural overview of the major segments of the client application, including its rudimentary communication with the server. UML arrow notation is used to represent aggregation and dependency as usual.

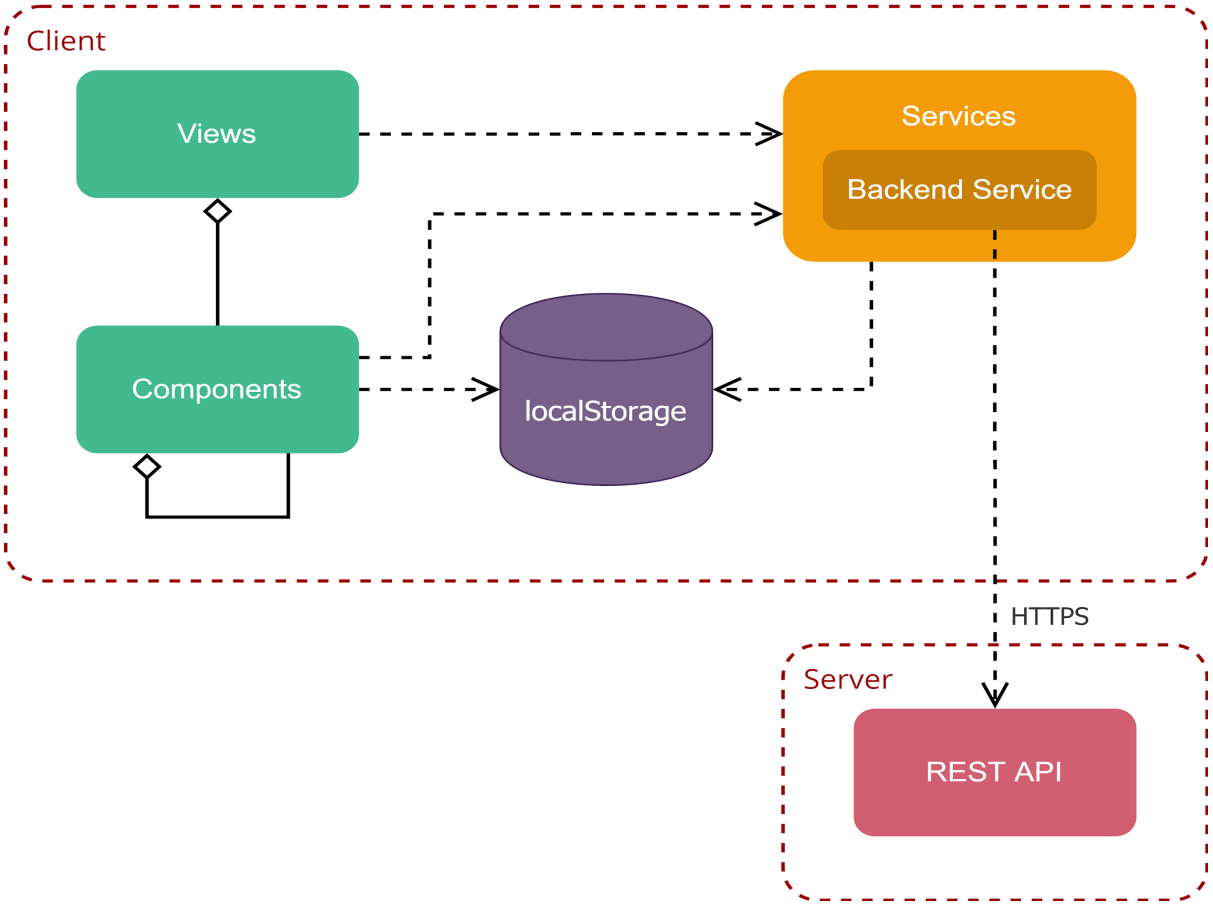

Figure 4 : *Diagram representing the structure of the client application*

Testing

This section describes the rigorous evaluation phase of the project. Each component of the source code was tested in a regular and constant fashion, individually as well as together.

For backend, code written followed a very much test driven development scheme where many tests were written using JUnit Jupiter so that after refactoring, code could be automatically unit tested to ensure functionality was still as intended. For this reason, an extensive amount of unit testing can be found in the back-end/src/test/java/ directory as this is the Gradle default. Tests can be run with Gradle test command from the build.gradle file's location in /back-end/.

Alongside Gradle and JUnit Jupiter, some of the more complex classes required database connections through Hibernate. For Manager and Controller classes, local hosted MySQL/MariaDB databases were set up through the following configuration files:

```
test/resources/testhibernate.cfg
test/resources/hibernate.properties
main/resources/application.yml
```

This allowed automated tests to connect to a separate test database to conduct CRUD operations, database constraint checks for leaky connections etc. Controllers can be tested by hand using Postman to simulate JSON Bodies sent into the REST API endpoints too - but will run using the main/resources files (e.g. POST localhost:8080/appinfo/ (discussed in the next section), but also has automated tests where Micronaut HTTPRequest objects are created and exchanged through a HTTPClient object. This tested that all our REST API endpoints were functional as expected. For those without a localhost, a branch from the github repository has been provided with a deployed testing database so that tests can still be run. See the developer's manual for information on how to do this.

| Code Coverage Statistics - Jacoco test report

back-end

Element	Missed Instructions	Cov.	Missed Branches	Cov.	Missed	Cxty	Missed	Lines	Missed	Methods	Missed	Classes
main.java.com.projectBackEnd		80%		55%	16	51	25	133	5	32	1	7
main.java.com.projectBackEnd.Services.AppInfo		77%		83%	3	25	10	52	2	22	0	4
main.java.com.projectBackEnd.Services.User.Hibernate		97%		97%	3	74	4	143	1	38	0	5
main.java.com.projectBackEnd.Services.User.Micronaut		94%		100%	0	45	10	124	0	37	0	5
main.java.com.projectBackEnd.Services.Image		96%		87%	4	38	6	70	0	22	0	3
main.java.com.projectBackEnd.Services.Page.Micronaut		97%		100%	0	38	4	87	0	32	0	4
main.java.com.projectBackEnd.Services.User.Hibernate.Exceptions		85%		n/a	1	7	2	14	1	7	1	7
main.java.com.projectBackEnd.Services.News.Hibernate		100%		100%	0	55	0	97	0	37	0	2
main.java.com.projectBackEnd.Services.Page.Hibernate		100%		100%	0	38	0	73	0	28	0	2
main.java.com.projectBackEnd.Services.Session		100%		93%	2	33	0	61	0	17	0	3
main.java.com.projectBackEnd.Services.News.Micronaut		100%		100%	0	28	0	62	0	25	0	3
main.java.com.projectBackEnd.Services.Site.Hibernate		100%		100%	0	27	0	49	0	20	0	2
main.java.com.projectBackEnd.Services.Medicine.Micronaut		100%		100%	0	24	0	38	0	17	0	3
main.java.com.projectBackEnd.Services.ResetLinks		100%		94%	1	23	0	48	0	14	0	3
main.java.com.projectBackEnd.Services.Site.Micronaut		100%		100%	0	20	0	41	0	17	0	3
main.java.com.projectBackEnd.Services.Medicine.Hibernate		100%		100%	0	20	0	32	0	15	0	2
Total	183 of 4,366	95%	27 of 332	91%	30	546	61	1,124	9	380	2	58

The overall test coverage is quite high: however, all of the missed instructions are justifiable. For example, some branches are inaccessible by regular means as they would require a change in design and code encapsulation (e.g., adding explicit public constructors).

HibernateUtility has lower coverage as the errors in the catches are caused by bad database faults or faults in the session factory. These are very rare when using a functional database and are thus difficult to test: they are only written as safety measures to ensure sessions are still shut down correctly to avoid trailing thread connection leaks to the database. There is also an imported test class which ensures none of these leaks persist. Should leaks persist, these are offcuts and small due to database lag - rerunning will remove them and they will timeout on their own. These are small and can be ignored as are infrequent depending on database connection speeds.

Other classes have inaccessible exceptions which can be very difficult to throw, like sending mail going wrong due to server connection, UTF-8 Encoding errors and interrupted thread sleeping. There is also code for random Image name generation and random session token generators that ensures they can never be duplicate names, however the chances of randomly generating a used string is too low and cannot be tested feasibly considering the number of different possibilities.

Some exceptions are thrown due to bad statically set file locations - this will also never occur in the test environment since JUnit Jupiter test classes share a static class between them all per run - and location is set upon static initialization with defaults. We did try to write a test class which unset these statics to test, but if any other class ran first, it would set up the statics correctly and the order of tests is unpredictable. On the server, where directories are unknown, private methods such as createFile() in AppInfo are run (and successfully, seen by deployment). These ensure IOExceptions are not thrown by these classes as the createFile method in the supporting class will trigger ensuring the relevant directories can be found.

Team Organisation

In this section, the agile team and each team member's role is described.

The team is constituted of eight members, with varying abilities in the technologies used. Our strength is the diversity of familiarity in different technologies; front-end, web applications, Javascript frameworks, Java and/or back-end. This allowed us to easily separate the roles and distribute tasks to each team member: we formed two groups, each specialised in back-end or in front-end.

Front-end group	Andy CIUFFO	[**Front-end Manager**]
	Callum WATKINS	
	Yicheng ZHAN	
	Ziyang CHEN	*Polyvalent*
Back-end group	Marek GRZESIUK	[**Back-end Manager**]
	Mohammad ISLAM	[**Tech Leader**] *Polyvalent*
	Kerim YALCHIN,	
	Jeanne BEYAZIAN	[**Team Leader**]

We defined responsibilities for sub-teams leaders based on technical abilities whereas the team leader had a role to maintain team cohesion and effective communication. As we were following agile methodologies, we kept some flexibility and had polyvalent members who were happy to switch from a group to another if needed. We also had a member in charge of reviewing, cleaning and maintaining high standards of code all along the project.

The Team Leader was responsible for :

- Liaison between team and stakeholders.
- Team organisational duties: setting up meetings, regularly reporting team progress to the team and meeting summaries, ensuring communication for the whole team.
- Conflict resolution : provide assistance to individual team members.
- Milestones deadlines.

The Front-end and Back-end Managers were responsible for :

- Ensuring communication within their groups and guiding group discussions.
- Technical support of individual members (in labs, by message or screenshare).
- Measuring their group's progress and reporting to the team leader.
- Deadlines for the tasks taken care of by their group.

The Tech Leader was responsible for:

- Overall knowledge of the source code (both in back-end and front-end)
- Most experienced member that would advise team managers for the next technical steps
- Guide other team members with the source code

Additional information_ _

Other resources and information about the project.

| Covid difficulties

We encountered a few difficulties communicating with our client in the last few weeks of the project. Our final meeting at King's College Hospital got cancelled as our client had new responsibilities due to the spread of COVID-19. We chose to minimise our contact with them as the times were difficult and they could hardly spend time answering emails. We could not present our fully functional product and get the last feedback on our work as we had planned.

Two of our team members had to travel back to their home in China during the pandemic. This caused a lot of issues for them as they were lacking a good wifi connection and had to use a VPN in order to use our team group chat. The time difference also made it difficult to plan meetings at a time when everyone was available. Fortunately, they were dedicated and accepted to come online at late hours to attend team meetings.

| Gantt chart used and updated by the team leader : **SEG [MAJOR] Gantt Chart**