ICBC Flex Work

Internal Design Document

February 13, 2020
Version 0.2

Team Flex:
Srijon Saha
Yifan Wei（Kevin）
John Zou
Charlie Chen
Ravina Gill
Linh Phan

Introduction

This document will outline how the Project Flex will be built. This will be achieved by defining the programming, production, and test environments, the software architecture, data design, API design, any complex algorithms, and notable tradeoffs and risks. This document will also capture system availability (including single point of failure), capacity, performance, scalability, security, maintenance, and UI design and screen mockups.

Programming Environment

Technology Stack

Front End: React (version 16.8), JavaScript (version ES8)
Back End:
Database: MySQL version 8.0
Web API: Node.js version 12.14.1 (including npm version 6.13.4), Express.js version 4.17.1
IDE: JetBrains WebStorm version 2019.3 and Visual Studio Code version 1.42
Source Control System: GitLab
UML Tool: Draw.io

Production and Test Environments

Backend

The server will be deployed on AWS EC2. We will use CentOS Linux distribution.

Frontend

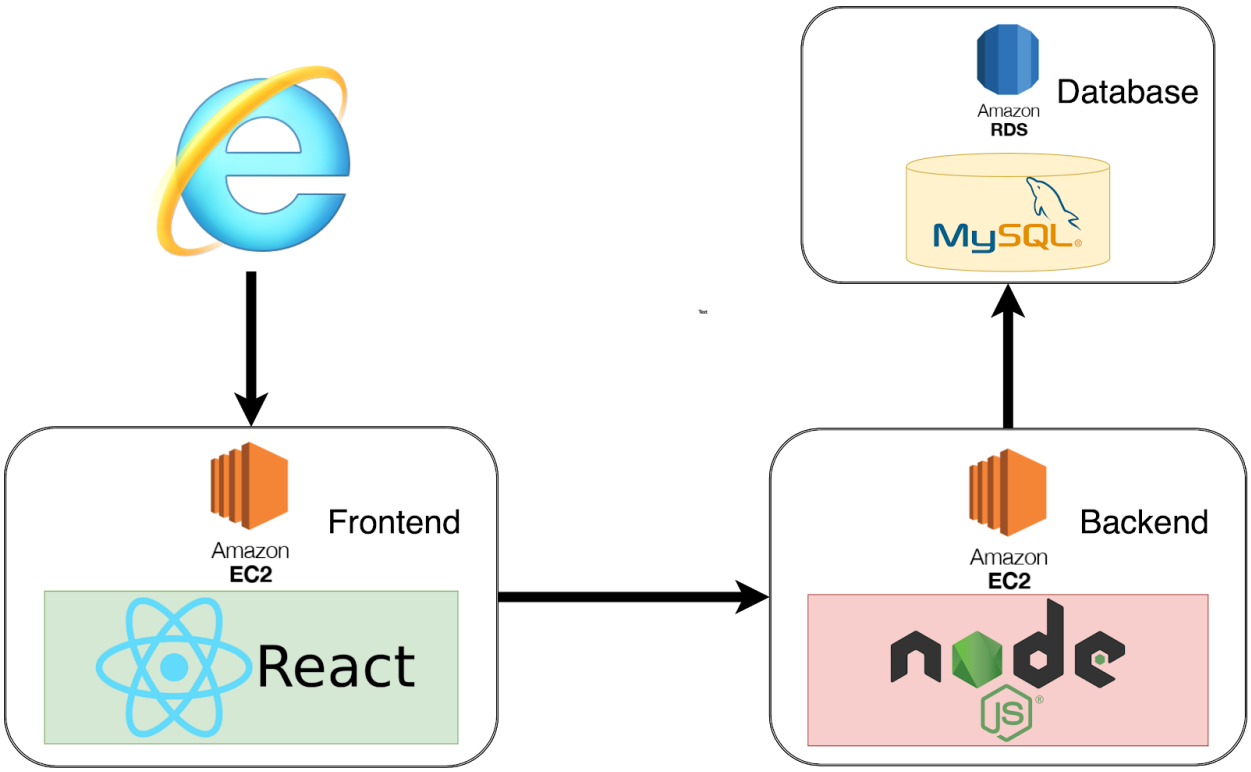
The React frontend will be deployed on AWS EC2. We will use CentOS Linux distribution.

Database

The MySQL database will be deployed on Amazon RDS.

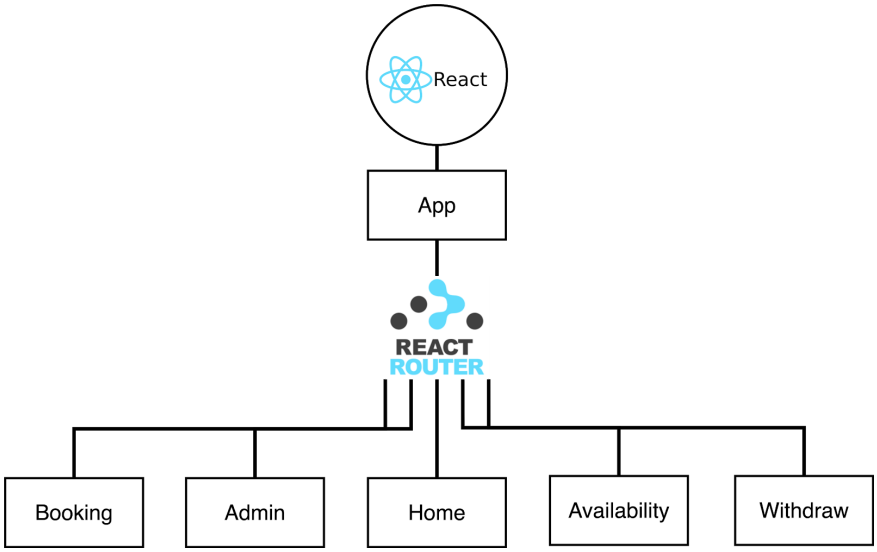
Software Architecture

The architecture of our application can be visualized by the following architecture diagram. The client end user will use Internet Explorer 11 to access our React-based frontend application that will be running on an Amazon EC2 server. The frontend will communicate with the backend API via HTTP requests to retrieve data related to our application. The Node.js backend will also be hosted on an Amazon EC2 server and will communicate with a MySQL database via the MySQL connector and it will be hosted on Amazon RDS.



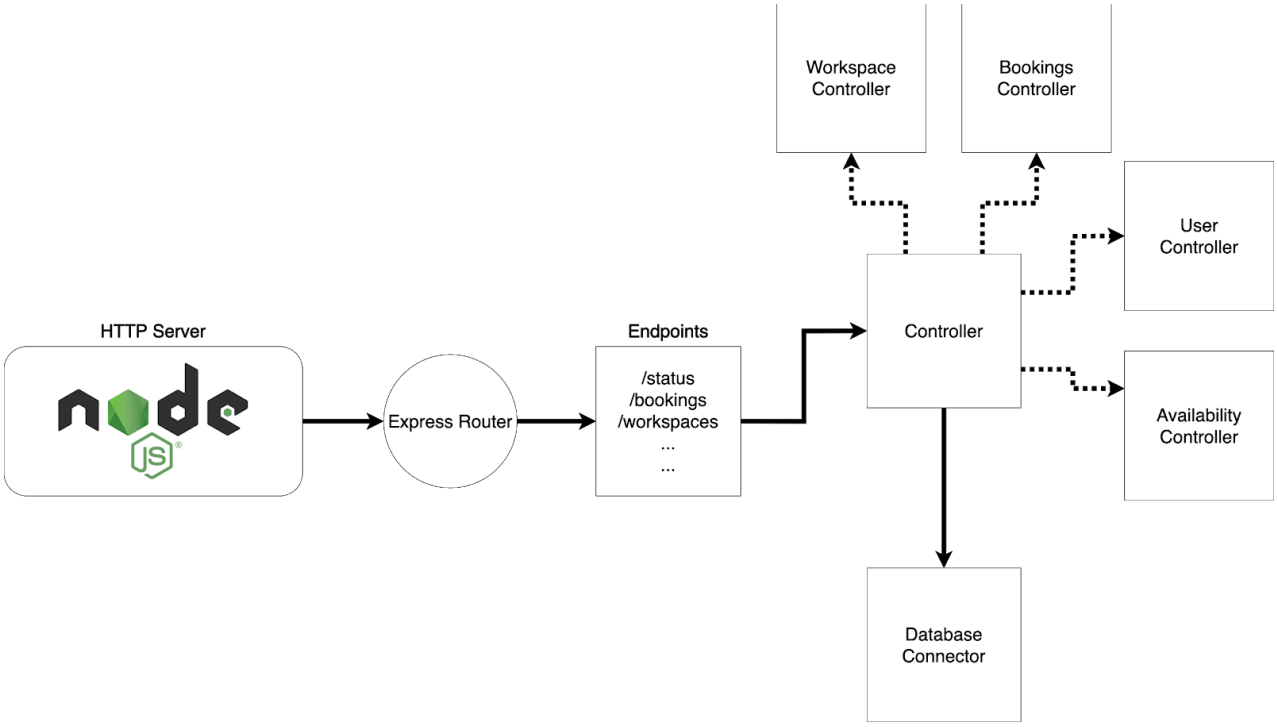
Frontend Architecture

The diagram below illustrates the components in our React app. The five main React components will be Home, Availability, Booking, Withdraw, and Admin. The App component will route to each of these components via the React Router based on the browser URL. Each of these components will be responsible handling the following:
Home: The home page that will show buttons/navigation links to the rest of the components.
Availability: Manage the UI of searching and registering a workspace for booking.
Booking: Manage the UI of the user searching for availability and placing a booking.
Withdraw: Manage the UI of user withdrawing an availability for booking under a staff id.
Admin: Manage the password protected interface to admin page that allows them to edit/delete existing entries in the database.



Backend Architecture

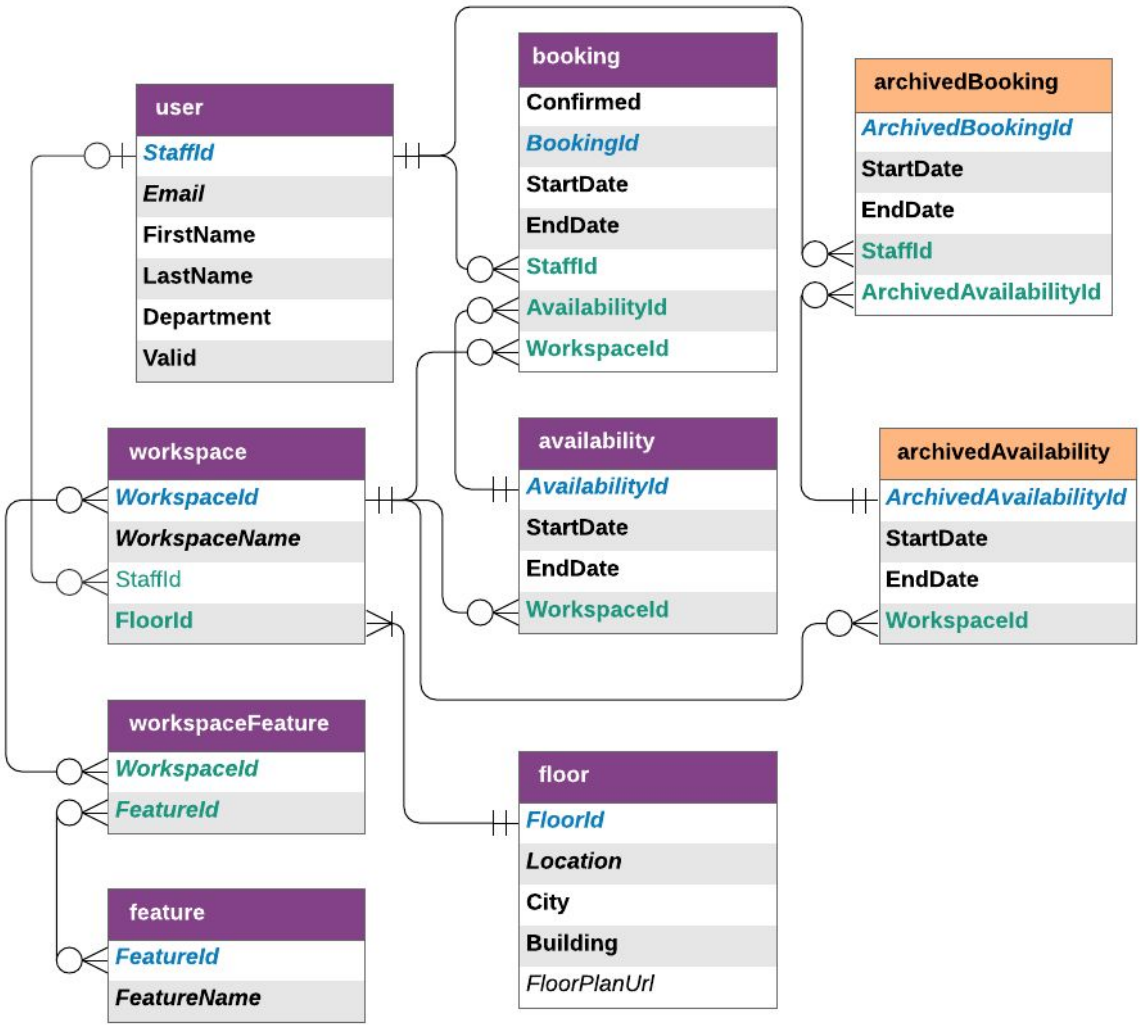
The backend provides create, read, update, and delete access to the database for the frontend React application. The backend will provide a REST API with endpoints for running specific queries to access the database. When a request is sent to a specific endpoint, a handler function from the respective controller (workspace, booking, user, or availability) will run a SQL query via the MySQL connector and return the results of the query as JSON for the frontend to parse and render. Some endpoints will require authorization for access such as endpoints that are concerned with deletion and editing of workspaces, users and can only be performed by the administrator.



Data Design

The design of our database will consist of the following tables and columns with the relationships shown clearly in ubiquitous crow's foot notation.

Flex Work Entity Relationship Diagram



Legend:
Bold = Mandatory, *Italics = Unique,* ***Blue = Primary Key***, Green = Foreign Key
Purple Entity Table Heading = For Normal Use Cases
Orange Entity Table Heading = For Archiving

Details of DB Entities

Legend:
Bold = Mandatory, *Italics = Unique,* ***Blue = Primary Key***, Green = Foreign Key

Table (Entity)	Columns	Type
user	***StaffId*** ***Email*** **FirstName** **LastName** Department **Valid**	Integer Primary Key Varchar Varchar Varchar Varchar Boolean
workspace	***WorkspaceId*** ***WorkspaceName*** StaffId **FloorId**	Int Primary Key Varchar Int Foreign Key Int Foreign Key
feature	***FeatureId*** ***FeatureName***	Int Primary Key Varchar
workspaceFeature	***WorkspaceId*** ***FeatureId***	Int Foreign Key / Primary Key Int Foreign Key / Primary Key
booking	***BookingId*** **Confirmed** **StartDate** **EndDate** **StaffId** **AvailabilityId** **WorkspaceId**	Int Primary Key Boolean Date Date Int Foreign Key Int Foreign Key Int Foreign Key
availability	***AvailabilityId*** **StartDate** **EndDate** **WorkspaceId**	Int Primary Key Date Date Int Foreign Key
floor	***FloorId*** ***Location*** **City** **Building** *FloorPlanUrl*	Int Primary Key Varchar Varchar Varchar Varchar

(Database tables, continued)

Table	Columns	Type
archivedBooking	***ArchivedBookingId*** **StartDate** **EndDate** **StaffId** **ArchivedAvailabilityId**	Int Primary Key Date Date Int Int Foreign Key
archivedAvailability	***ArchivedAvailabilityId*** **StartDate** **EndDate** **WorkspaceId**	Int Primary Key Date Date Int Foreign Key

Notes:

Archive Tables:

We will archive Availabilities and Bookings whose EndDate has passed by running a DBA script. This script will run automatically at every midnight.

The DBA script will move all Availabilities from the availability table whose EndDate is before current date, to archivedAvailability, and move all of its bookings to archivedBooking, setting their archivedAvailabilityId foreign key to that of the copied Availability it belongs to.

Normalization:

Our entity tables are mostly normalized. The exception is for unique identifiers/primary keys: we decided to use integer for the primary key for every entity because that is the industry norm (for performance and maintenance reasons) even if we enforce other fields to be unique (such as user.Email).

Cardinality:

The relationship decisions are for flexibility of business needs.

For example, the user-workspace relationship is one-to-many, optional on both sides (zero-to-one, to zero-to-many): this is to allow the case of a single employee owning multiple offices. (Of course, it is necessary to allow for the situation of an employee who hasn't been assigned a permanent desk, and a desk to have no permanent employee. Rationale for the cardinality of each relationship is similar and based on business needs.

API Design

The main purpose of the Node Back End will be to provide a REST API for the Front End (and other potential mobile applications, such as mobile applications).

See Appendix I for detailed intended usage and specification of these end points with regards to Use Cases, and in particular, parameters for GET.

Backend API Endpoints

Resource	**GET**	**POST**	**PUT**	**DELETE**
/bookings	Returns bookings	Creates a new booking	N/A	N/A
/bookings/:id	N/A	N/A	Update a booking	Delete a booking
/availabilities	Returns availabilities and their bookings	Create a new availability	N/A	N/A
/lockWorkspace	N/A	Lock availability with certain timeframe	N/A	N/A
/availabilities/:id	N/A	N/A	N/A	Delete an availability
/workspaces	Return workspaces	Create an workspace	N/A	
/workspaces/:id	N/A		Update workspace	Delete workspace
/users	Return a user	Add a user	Update user	Delete a user
/status	Show the API status	N/A	N/A	N/A
/backups	Returns a backup of the database (stretch goal)	Restore the database from a backup	N/A	N/A

/adminLogin	N/A	Start a session	N/A	End a session
/floors	Get floor plan based on location	Create a floor	Edit a floor	Delete a floor
/floorplan	N/A	Upload floor plan JPG	N/A	N/A

Client Endpoints

The following endpoints are provided to the client end user and is accessible via the browser to interact with the Flex Work application:

Endpoint	**Description**
/	Homepage providing access to the other interfaces (i.e. booking, registering, admin)
/availabilities	From to search and mark a workspace for available for booking by other employees
/bookings	Form to search workspaces and place a booking for workspace
/withdraw	Shows all the bookings made a user once the staff id is entered. Also gives the user an interface to withdraw any bookings made by the staff id.
/admin	Administrator dashboard that allows them to create, update, or delete bookings and locations. This page is initially protected till the administrator password is provided.

Algorithms

When we search for available workspace with start date and end date, we need a way to identify availabilities with bookings made on it. For example, if A made his workspace available from Jan 1 to Jan 10, B made a booking from Jan 1 to Jan 9. Then C wants to search for an availability from Jan 1 to Jan 3. This workspace will still appear in the search result because our SQL query will be all availabilities from Jan 1 to Jan 3. To handle this situation, our system will split availabilities after a booking has been made. Combine the availability if booking is canceled. In the above situation, after B placed booking from Jan 1 to Jan 9, we will split this

availability into 2. One from Jan 1 to Jan 9 with bookingId, one from Jan 9 to Jan 10 without bookingId. That way we can distinguish availability with and without booking made on it. If B cancels the booking, we should join the two availabilities records into one. Detailed information is described below.

Split Availabilities

When a booking has been made on an availability, we will need to split this availability into different records. Assume availability being made is from date “a_start” to “a_end”, booking is from date “b_start” to “b_end”. Since booking can only be made within the availability period, booking has to be at least one day, we have “a_start” <= “b_start” <= “b_end” <= “a_end. The split operation should split this availability record into at most 3 part, “a_start” to “b_start”, “b_start” to “b_end”, “b_end” to “a_end” (for each part if the start date equals to end date, that part should not be added). The record starts from “b_start” to “b_end” will have a BookingId assigned to it. Since the unbooked period does not have BookingId assigned, we can get the correct availability record when searching for available workspace.

Combine Availabilities

When a booking has been removed from an availabilities, we should combine separate availabilities into a concrete one. Assume we want to remove booking from date “b_start” to “b_end” on availability date “a_start” to “a_end”, we should join at most 3 availability records into one (“a_start” to “b_start”, “b_start” to “b_end”, “b_end” to “a_end”). Since every booking record has AvailabilityId, and availability record has workspaceId, we can find all availability on that workspaceId, then join them if (a) record does not have BookingId assigned to it. And (b) the start date of a record equals the end date of another record. Then we can safely delete the booking record.

Notable Trade-offs

Identify design tradeoff (ie: need to make certain design decisions based on time, quality, performance, security, or scope considerations

The design of the database schema is based on how users interact with this system and the assumptions we made, such as one ICBC staff can only have one permanent workspace, and it is reflected in our database structure --- the user table and workspace table has one to one relationship. When the user wants to put his/her workspace available for others to use, the user can put it in multiple periods. Therefore one workspace corresponds to multiple availability entries but one availability can only have one workspace, availability table and workspace table have many to one relation in our database design. Same idea goes with bookings and availability, multiple bookings can be made on one availability entry but one booking entry can

be linked to one availability entry. Since one user can make multiple bookings, one booking entry can only have one user, the user table and booking table has one to many relationships.

Considering the fact that this system should be able to handle 1000 live active bookings and up to 20 concurrent usage at peak time, we decided to use relational database in our design and deploy it on Amazon RDS. The back-end will be implemented using Node.js framework, deployed on Amazon EC2. Because it is relatively approachable for our team and fully capable of handling such usage.

The endpoint will return all the records that satisfy requirements, which can make the webpage really slow. Since we only have to handle 1000 live active bookings, this issue is trivial. If needed, we will make changes to our endpoint to break records into pages.

Notable Risks

Development Risks

As a result of the requirement of Internet Explorer 11 compatibility, we have a development risk, as most of our team uses Mac OS on our development machine. Further, React requires several additional configuration steps (including downgrading the default version of dependencies) before it can run on Internet Explorer 11. Both of these add additional complexity to our workflow (necessity of a virtual machine, additional configuration), and could slow/impede development. We might also need to take additional steps to ensure React Components work with IE11 -- many are incompatible. For example, some of the most common date-pickers, a key component of our front end UI, are incompatible with IE11. This may lead to additional development time as we find compatible components or write our own if we cannot find one.

We believe these to be necessary risks, as IE11 compatibility is a core requirement, and the benefits of the React framework far outweigh the cons.

Risk Tolerance for Stakeholder

Lost data would mean losing booking information. This is preventable with daily backup/snapshot of the SQL Database, which we will implement as part of the Back End as an automated task as a self-imposed stretch goal.

We are deploying our MVP on Amazon RDS free tier, which provides 20 GB of backup storage for both automated and manual database backups. This is enough to hold a full daily backup snapshot effectively indefinitely (at least a decade).

However, if used on ICBC internal servers, we recommend ICBC to have some back up medium available. ICBC may choose to only keep backups going back a week or a month, at their discretion.

For the admin interface, using HTTPS and an authentication token should make it difficult for anyone to obtain the admin password by network sniffing or man-in-the-middle attack. The admin password is a point of weakness, however, because it may be easily obtained by other means (such as social engineering). To circumvent this, we will not allow anyone, including the admin, to delete the daily backups, and this will serve as a way to mitigate the risk of someone malicious obtaining the admin password in the MVP. This also serves as a way to prevent an employee or anyone in the intranet to abuse the honor system.

In the final product that ICBC would deploy, the admin password will be replaced with authentication/authorization provided by Active Directory and would no longer be a point of weakness.

Sensitive information -- we store some sensitive information, such as an employee's full name and email address, on Amazon RDS, which is encrypted. The Back End REST API endpoints, which is the only way to access this information, will only be accessible from within the ICBC intranet. In any case, the leakage of names, email addresses, and booking information of the employees will not involve a great cost or value.

System Availability

Amazon EC2 server ensuring the proper interaction between users and backend work plays the most important role in the systems.So this server is identified as the single point of failure.

Performance

The system is supposed to handle concurrent usage of up to 20 users as well as hold the reservation for 20 mins. Exceeding time limit or closing browser will release the lock for reservation.

Scalability

The system is expected to handle about 1000 peak live active bookings in the system, but average volumes will most likely be in the 300 range. We will also be storing the floor plan images. With thousands of entries on each category of booking, user, and office, the maximum

storage anticipated will be not more than a few megabytes. The system is therefore very scalable, even though there is no anticipated need for massive scaling.

Security

For participants who book or release the offices, we use an honor system implemented by ICBC. So it is possible that one user having access to the system might be able to make a booking for others. And for administrators who manage all the bookings , we use a simple password for verification because people other than that role are not supposed to edit or delete bookings .

Maintenance

Upgrades

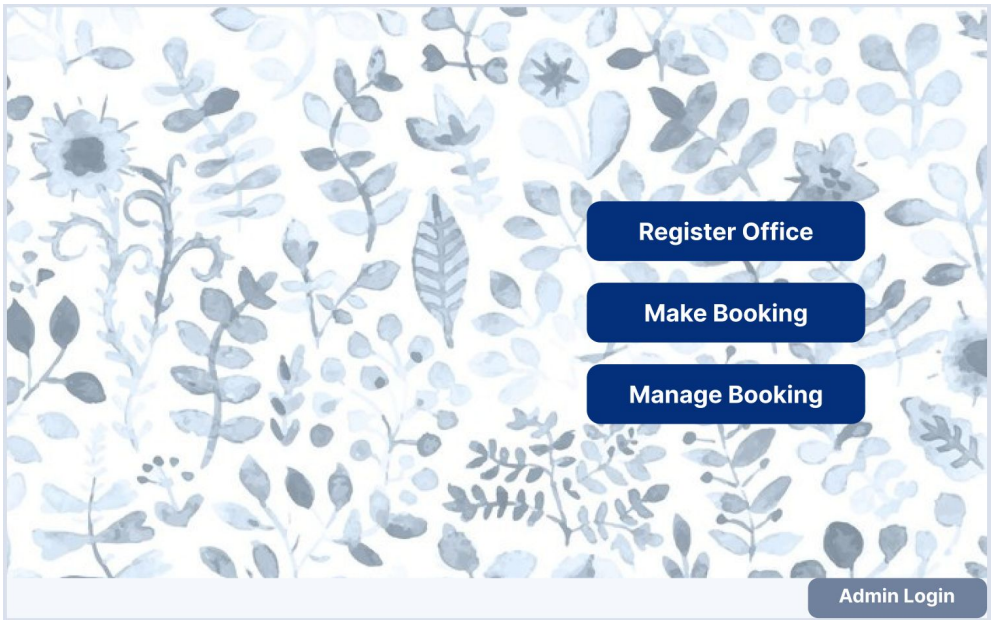
ICBC will have access to source code, which they can then modify as required for updates and deploy.

Database

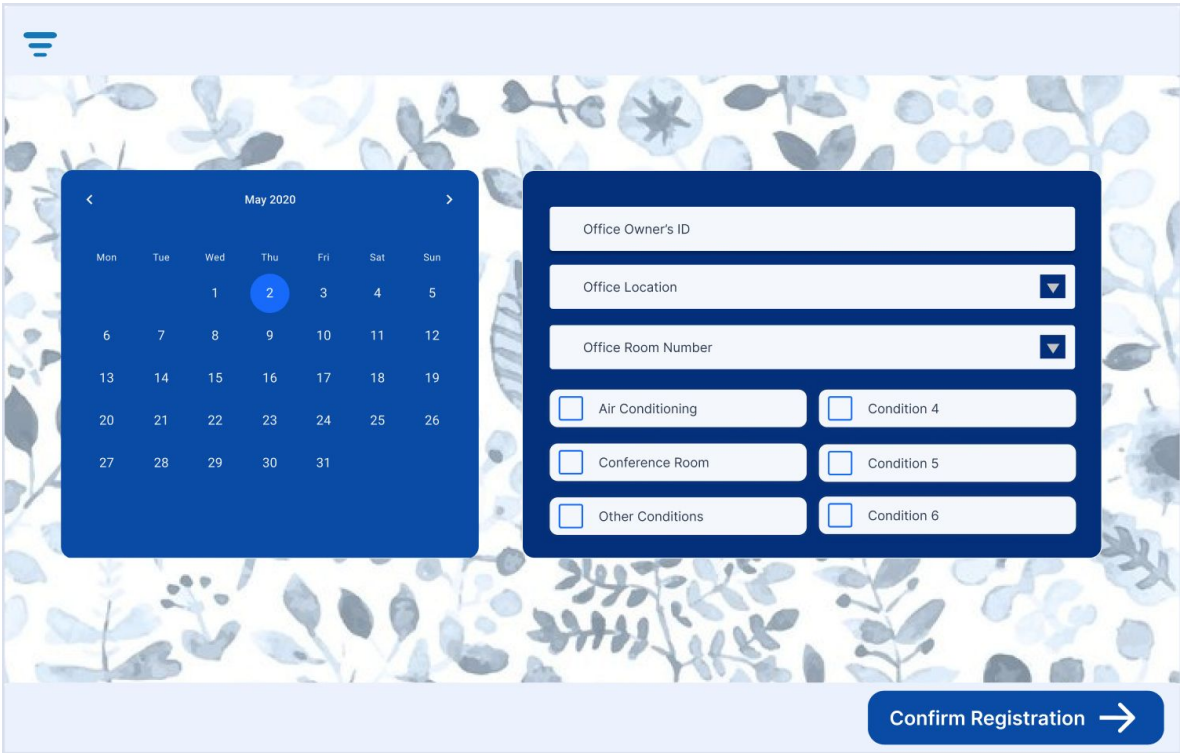
We will implement automated daily backup of the SQL Database, along with automated monthly retiring of old bookings/availabilities into the archive tables.
ICBC will not need to manually perform these actions. However, should they wish to modify this process, they may do so directly through the source code of our Back End.

UI Design and Screen Mockups

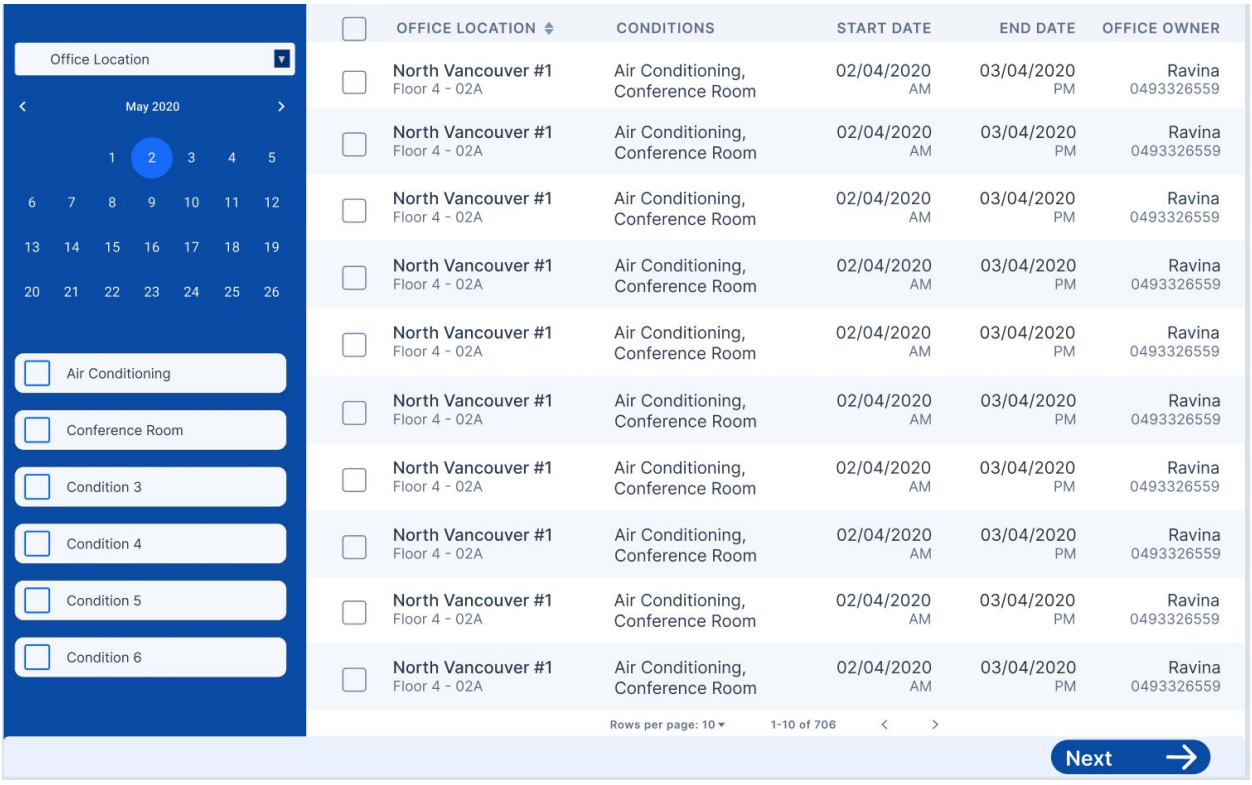
Homepage

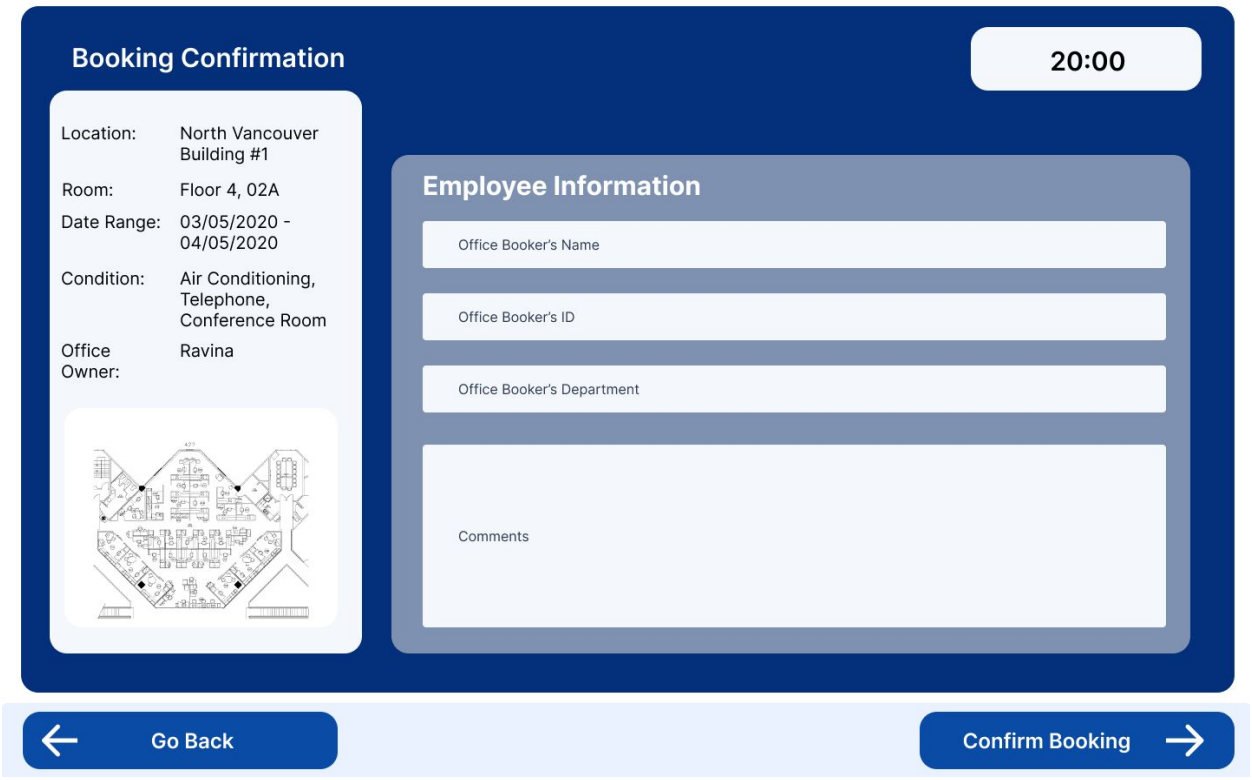


Registering Workspaces for Availability

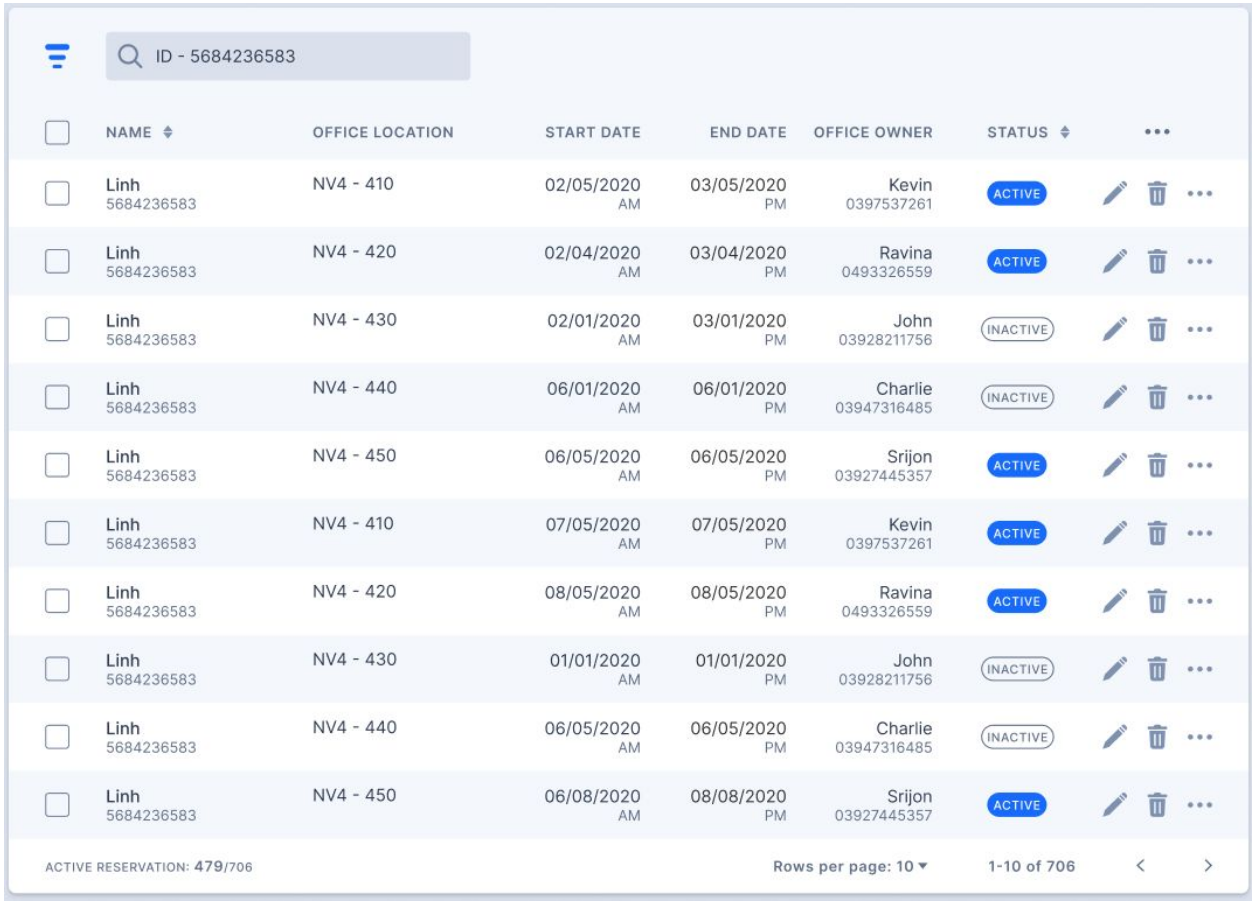


Search and placing a workspace booking





Listing existing bookings to edit or withdraw



Appendix I - The Back End REST API as it relates to Use Cases

1. User (ICBC Employee)
    1.1. User searches for all available workspaces with filter
    1.2. User enters the confirmation page with an available workspace
    1.3. User confirmed booking in confirmation page
    1.4 User wants to check all the bookings made
    1.5 Desk-given user wants to check all the bookings made on the availability
    1.6. User wants to delete an upcoming booking
    1.7. User wants to mark workspace as available
    1.8. User wants to cancel availability
2. Admin (ICBC Facility Administrator)
    2.1. Admin login
    2.2. Admin add user
    2.3. Admin delete user
    2.4. Admin add floor plan
    2.5. Show floor plan based on filter
    2.6. Admin update single workspace information (assign it to someone else)
    2.7. Admin delete workspace
    2.8. Show all bookings made by filter
    2.9. Update booking
    2.10. Admin delete booking
    2.11. Show all availabilities
    2.12. Delete availabilities

. 3. DBA script(runs automatically)
    3.1 Offload non-active bookings into an ‘archive’ table

If any HTTP request is made with invalid parameters/body values/missing information etc. then respond with 400 Bad Request.

1.1 <u>User searches for all available workspaces with filter</u>

GET **/availabilities**

Mandatory params: start date, end date,

Optional params: location, city, floor, building, conference phone, TV, private

Response: 200 OK:

- Join of availability-workspace-location-user[1] filtered by the params (call this AWU)
- Respond with array of Availabilities (AWU rows) with an extra array field holding the associated BUs for each AWU

Front end will check if the user specified the start date and end date. If not, frontend will reject users' requests. The back end will get all available workspaces that satisfy the filter.

[1] we need to join with the user table to show who owns the workspace.

1.2 User enters the confirmation page with an available workspace

Post /lockWorkspace

Mandatory params: AvailabilityId, StaffId, Start time, End time

Response: 200 OK with bookingId

- If the Start time and End time of booking is within the available time period of corresponding AvailabilityId

  403 Forbidden: if the booking is not successful.

The backend is going to insert a record in the bookings table. This record will leave the confirmed field false and return the bookingId. This record will be deleted in 20 min if the user does not confirm booking[1].

1.3 User confirmed booking in confirmation page

POST /bookings

Mandatory param: BookingId

Response: 200 OK

403 Forbidden: if the booking is not possible

Backend mark booking confirmed by changing the confirmed field for that BookingId to true. The backend will also stop the timer.Email notification should be sent to both booker and workspace owner.

1.4 User wants to check all the bookings made

GET **/bookings**

User wants to check all the bookings made and do modifications afterwards.

Mandatory param: StaffId

Response: 200 OK

- SQL query StaffId on joint of user-bookings-availability-workspace-floor table. Return all information contained in JSON format group by bookings.

  403 Forbidden: if there is an error occurred.

1.5 Desk-given user wants to check all the bookings made on the availability

GET **/availabilities**

Mandatory param: StaffId

Need two different actions:

Response: 200 OK

- SQL query StaffId on the joint of user-bookings-availability-workspace-floor table. Return all information contained in JSON format grouped by availability.

  403 Forbidden: if there is an occurred.

The frontend should show all availabilities made by this user. If a user wants to check who made bookings on his workspace, the user can click on that availability record, the front end can filter by availabilityId.

[1] a join Availabilities operation described in Algorithm section should be performed

1.6 User wants to delete an upcoming booking

DELETE /bookings/:id

Mandatory param: bookingId

Response: 200 OK

- The records on the bookings table are successfully deleted.

403 an error occurred.

Frontend will only allow users to make deletions on upcoming booking. This operation will trigger a joint availability record described in the Algorithm section. Email notification should be sent to both booker and workspace owner.

1.7 User wants to mark workspace as available

POST /availabilities with mandatory body values: start date, end date, workspace ID

Response:

200 OK available record added successfully in availability table.

403 Forbidden: if there is already an availability with conflicting start and/or end date on this workspace

Frontend will only allow users to enter a start date greater than today. Email notification should be sent to the workspace owner.

1.8 User wants to cancel availability

DELETE /availabilities/:id

The front end has Availability ID from Use Case 1.3 subcase 2

Bookings made on that availability are deleted as well, email notification should be sent.

Response:

200 OK

- Availability record deleted successfully
- AND Bookings made on that availability are deleted successfully.

403 Forbidden: if doesn't exist

2.1 Admin login (red means associated with authorization)

POST /adminLogin (NEED HTTPS)

Mandatory body value: the actual password (hence this method needs HTTPS)

Response:

200 OK, with token. The token will be included in the body of every admin request.

403 Forbidden: wrong pass

2.1.1 Admin logout: DELETE /adminLogin/:token

2.2 Admin Add User

POST /users

Mandatory body values: Name, Email

Optional parameter: Workplace ID (an employee who hasn't been assigned a desk should still be able to book a desk)

Response:

200 OK mark field Valid true in user table

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

403 Forbidden: Conflicting email or desk already belongs to another user

2.3 Admin Delete User

Admin needs to be able to see users.

GET /users

Response:

200 OK: An array of users or an empty array

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response.

The front end will further allow admin to filter by any combination of user fields.

DELETE /users/:id

Response:

200 OK change Valid field in user table to false.

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

403 Forbidden (user doesn't exist)

Deleting a user does NOT delete their availability because availabilities are owned by workspace, not user of the workspace. The bookings made by that user should be deleted. The delete booking operation should be same as use case 1.6

2.4 Admin Add Floor Plan,

POST /floors

To add a floor plan, admin should upload an Excel/csv table with all the information. This endpoint should parse information in the table then add to the workspace and floor table.

Response:

200 OK with floorId

403 an error occurred

POST /floorplan
Mandatory param: floorId
This endpoint is used to upload floor plan JPG image for corresponding floorId. The backend should store image in the server and store the path to image to database.

Response:

200 OK if floor plan JPG stored successfully

403 an error occurred

2.5 Show floor plan based on filter

GET /floors

Mandatory param: location

User/Admin can see floor plan by making GET request to backend. Backend will query floors table with unique entity location to find path to floor plan JPG stored in server, then send back the image to frontend.

Response:

200 OK with floor plan JPG

403 an error occurred

2.6 Admin update single workspace information (assign it to someone else)

2.6.1 To see workplaces, Admin will only be able to search by location.

GET /workspaces

Mandatory parameter: Location

Response:

200 OK: join of Workspace-User-Location

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

PUT /workspaces/:id

Mandatory body values: the entire workspace object with all fields mandatory (the front end will add default values for missing items) The back end can expect all fields to have be present.

Response:

200 OK

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

403 Forbidden (workspace doesn't exist)

2.7 Admin delete workspace

Front End will have workspace ID via Use Case 2.6.1.

DELETE /workspaces/:id

All associated availabilities and their bookings records are deleted in availabilities table and bookings table. Respective users will be emailed.

Response:

200 OK

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

403 Forbidden (workspace doesn't exist)

2.8 Show all bookings made by filter

Even though this is an admin use case, it does not need authorization.

I believe the endpoints from Use Case 1.1 and 1.3 are enough.

2.9 Update booking

Only change start/end date, do not change availability ID or user ID.

Front End will know booking ID via Use Case 2.8 / 1.1 / 1.3.

PUT /bookings/:id

Mandatory request body values: the new start and end dates.

Response:

200 OK

401 Unauthorized (missing, wrong, or expired security token) – Front end will show admin login screen in response

403 Forbidden (new dates conflict with availability / current bookings, or booking ID doesn't exist)

This use case should be handled by 1.3, 1.6 because split and join availability operation is required.

2.10 – 2.12 – end points provided for the user interface are sufficient.

3.1 Offload non-active bookings into an 'archive' table

The backend will have a DBA script that moves all the records from the bookings table to the archievedBookings table, if the end date has passed. It will also moves all the records from availabilities table to the archievedAvailabilities table. This script will run automatically at every midnight.