Project Type: Greenfield in a novel domain

Use Cases:

Use Case	Description
UC-1: Store catch information	Casual and Professional fishermen can store the data of fish they cache in the database. They can choose whether or not to share this data with other users on the platform.
UC-2: Extrapolate and analyze fish information	Professional fishermen can utilize their catch data to determine when the best time to fish is, the best locations to fish at, and the best bait to use for a specific type of fish.
UC-3: General wildlife information storage	All users can utilize the platform to document wildlife sightings other than fish, such as a sighting of a rare/near-extinct animal.
UC-4: Sharing caches with other users	Users can elect to share their catch information with other users on the platform on a case by case basis.

Non-Functional Requirements:

Quality Attributes:

ID	Quality Attribute	Scenario	Associated Use Case
QA-1	Security/Privacy	The user should be able to select whether or not to share their catches with other users. This preference can be edited at any time.	UC 4
QA-2	Security/Privacy	The user's entered personal information should be protected and hidden at all times.	All
QA-3	Availability	The user should be able to access their fishing information at any time and should	UC 1-3

		have offline access if necessary.	
QA-4	Performance	Users should be able to perform normal operations on the database quickly, even when many other users are connected.	All
QA-5	Performance	The database should consistently update with new information every 5 minutes.	UC 1-3
QA-6	Scalability	The database will store catch information for extended periods of time. Allows for viewing of previous year’s catches for data extrapolation.	All
QA-7	Deployability	The system will be deployed into production at all times, with separate environments for development and testing to ensure that the service is always running.	All

Constraints

ID	Constraint
CON-1	Large amounts of data must be stored in the database in the event that many users register for the service.
CON-2	The database that stores the fishing information is online only, so it cannot be accessed without an internet connection.
CON-3	Information that has been chosen to be shared must be stored separately from private information.

Architectural Concerns

ID	Concern
CRN-1	Getting the database up and running, along with allowing CRUD operations for multiple users.
CRN-2	Potential for offline storage/access of fishing data.
CRN-3	Entering fishing information into the database manually, does not lend well to expansion.

ADD Iteration 1

Step 1. Review Inputs

- Primary Quality Attribute (QA) scenarios prioritized

Design Purpose: This is a greenfield system in a relatively novel domain. We will design the system following a plan-driven process to ensure that everything is properly planned out. The beginning architectural design will satisfy the basic operations of the application.

Primary UCs: UC 1 and 3

Primary Quality Attribute (QA) Scenarios Prioritized

Scenario ID	Importance to the Customer	Difficulty of Implementation According to the Architect
QA-1	High	Medium
QA-2	High	Medium
QA-3	Medium	High
QA-4	Medium	Medium
QA-5	High	Low
QA-6	High	Low
QA-7	High	Medium

Step 2. Establish iteration goal by selecting drivers

- Determine which QAs, CONs and CRNs to include as drivers

Goal: Basic database functionality and overall system structure

Refer to CRN-1

- CONs 1-3
- QA 1-2: Security/Privacy
- QA 3: Availability
- QA 4-5: Performance
- QA 6: Scalability
- QA 7: Deployability

Step 3. Choose one or more elements of the system to refine

- Initially, the entire system needs to be refined

Step 4. Choose one or more design concepts that satisfy the selected drivers

- Select a reference architecture

Centralized System

- General Information
    - Ideally, the project should be created as a centralized system, due to the main functionality relying on a main database where all the information is stored.
    - Information will be partitioned into different areas of the database depending on what it is, but will still be a part of the central database.
    - Argument for a decentralized system would be that multiple servers/access points to balance the load, but does not seem necessary here.

Considered Extension of the Centralized System

- Transparency
    - The application should appear as a single system to the user, with as little delay and ping as possible
- Openness
    - The software should be kept simple and easy to understand by using standardized protocols
- Scalability
    - The application can be scaled out with server size
    - Distribution of the application would come naturally with size (larger area-of-service)
    - All servers would essentially be duplicates, containing the same database information and general structure
- Security
    - User accounts will be kept secure with backup authentication methods to prevent data interception/theft
    - Local database will be kept in the event of DDOS or unauthorized modification/fabrication in the database, and this local database can be used to restore the main database if needed
- QoS
    - The service should be quick, even with many users connected at once and many entries stored in the database
- Failure Management

    - In the event that new bugs or service-breaking additions are created, the service can be rolled back to the previous working version

Considered Large Architectural Patterns

- Leader-Follower Architectural Pattern
    - Due to the nature of the service being centralized and reliant on a database, this architectural pattern does not seem suitable since there is no real time processing, and most service operations occur within the central database rather than on follower processors.
    - A potential extension could use follower processors to distribute the load across a large geographical area. However, each follower processor would need the same features as the central database rather than specifying operations to be performed by decentralized follower processors, which could be expensive.
- Two-Tier Client-Server Pattern
    - I would deploy either the thin or fat clients for the two-tier client-server architecture. The application does not have a high amount of processing, that mainly lies in the server database, so thin would be a good option for this. However, the fat client may provide easier access to implementing additional features despite the increased difficulty in managing the clients.
    - Between the thin and fat clients, the fat client seems like a better choice due to the enhanced local processing utilizing cached database information in situations where internet connectivity cannot be guaranteed. Despite the increase in processing overhead on the local device, there will not be very much data visualization so performance should not take too much of a hit.
- Multi-Tier Client-Server Pattern
    - I would not deploy the multi-tier client-server architectural style. Although the management of clients may be easier in comparison to the two-tier model, the separation of data does not seem worthwhile since it then increases the volatility of the data. Along with this, the potential for data caching and offline access is significantly lower.
- Distributed Components Architectural Pattern
    - The Distributed Components Architecture does not seem good for the project, since it would be splitting the database into multiple parts. Many features of the service are interlinked, so increasing access time or potentially denying it outright would be detrimental.
    - SOA also seems similarly too complex to implement.
- Peer-to-Peer Architectural Pattern
    - Peer-to-Peer does not seem like a good choice for the project since the database would ideally be centralized with the data storage, rather than each individual node/user having the entire database localized on their device.

Selected Large Architectural Pattern: Fat Two-Tier Client-Server Pattern

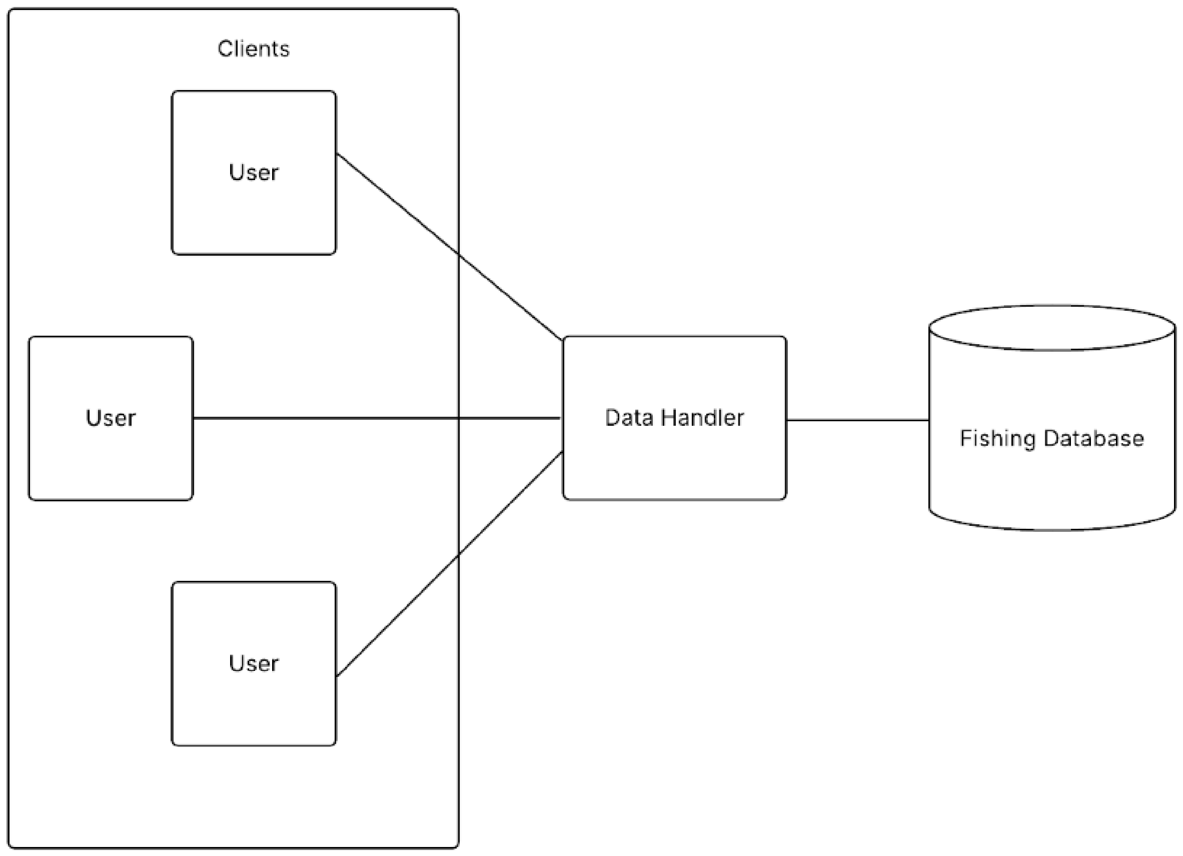
Step 5: Instantiate architectural elements, allocate responsibilities and define interfaces

- Select design decisions and determine rationale

Design Decision and Location	**Rationale**
Convert to a distributed system by utilizing the Two-Tier Client-Server architecture due to the long-term benefits	The Fat Two-Tier Client-Server Architectural pattern makes use of the available processing power on the device running the software, which allows for additional functionality that can be implemented in future iterations, such as data caching. This conversion was done due to the decision of offline access being undetermined. By converting to a distributed system, this would keep the option open in the future given that it would be a large boon to users.
The central database will act as the transaction server for information, and user devices will act as clients	In line with the Fat Two-Tier Client-Server Architecture, user devices will act as clients which will connect to the central database to receive or send information. This is associated with the following drivers: ● Database Storage (UC-1, UC-2, UC-3, QA-5, QA-6, QA-7, CON-1) ● Client View with CRUD Operations (QA-2, QA-4, CON-3)

Step 6. Sketch Views and Record Design Decisions

Two-Tier Client Server (Fat) Diagram



Element	**Responsibility**
Textual Data Storage	Textual information entered manually by the user. Can be for fishing catches along with other wildlife entries. Fishing data can be analyzed. This data will be stored in JSON format. (UC 1-3, QA-2, QA-6, CON-1, CON-3)
Image Storage	Images of popular formats such as .png can be uploaded and stored on a per-entry basis. This element is part of the Catch View. (UC1 and 3, QA-6, CON-1, CON-3)
Catch View	This view will show the user all of the information relating to the selected database entry. From here, the entry can be edited as desired. (UC 1-3, QA-4, CON-1)

Catch CRUD Operations	Create, Read, Update, and Delete operations. Users will be able to create new database entries, view their previously created database entries, update their previously created database entries, and delete their previously created database entries. Read, Update, and Delete operations are provided through the Catch View. (UC 1-3, QA-4, CON-1)
Drop-Down Menu Views	This element will comprise preprocessed drop-down menus which have the most common and popular fish types, locations, and bait types in the area. This allows for quick selection of attributes for data entries rather than manual entry. (UC 1-2)

Step 7. Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

Not Addressed	**Partially Addressed**	**Completely Addressed**	**Design Decisions Made During the Iteration**
	UC-1		Used the Fat Two-Tier Client-Server Architecture to allow users to connect to the database and store/view catch information. No offline access currently.
	UC-2		Textual Data Storage allows pro-fishermen to view important catch information to analyze and extrapolate data to better make decisions. No offline access currently.
	UC-3		Textual and Image Storage allows more general wildlife entries to be made. Only manual entry is available for this feature. No offline access currently.
UC-4			No relevant decisions made.
QA-1			No relevant decisions made.

		QA-2	Any personal information entered by the user will be stored securely on the server. Security features mentioned previously will keep information safe.
	QA-3		Although no offline access exists at this point, caching can be implemented in relation to the selected architecture (TTCS).
		QA-4	Due to the selected architecture, processing load is dependent on the client and not the server, therefore processing load should be well distributed geographically.
		QA-5	Due to the selected architecture, the database itself should not have much processing load relegated to it, therefore it will be speedy in its updates.
		QA-6	Text and Image Storage allows for long term data storage in the database.
	QA-7		Once the service enters production, a separate branch will be utilized for the implementation of new features.
	CON-1		Since the basic framework for the service has been created, users can store data privately.
CON-2			No relevant decisions made.
CON-3			No relevant decisions made.
		CRN-1	Users can create, read, update, and delete their own database entries.
CRN-2			No relevant decisions made.
	CRN-3		The adoption of the Fat Two-Tier Client-Server architecture means that new features and the expansion of current features can be implemented easier.

Iteration 2: Implementation of data caching and catch entry sharing

Step 2. Establish iteration goal by selecting drivers

Goal: Implementation of Key Features

Refer to CRN-2

Selected Drivers:
- CONs 2-3
- QA 1-2: Security/Privacy
- QA 3: Availability

Step 3. Choose one or more elements of the system to refine

The selected drivers, primarily pertaining to the implementation of the data caching and cache entry sharing features will be refined.

Step 4. Choose one or more design concepts that satisfy the selected drivers

A selection of architectural designs and styles were considered for the selected drivers.

- Model-View-Controller Architectural Pattern
  - The MVC does not seem like a good fit since there is only one way of viewing and interacting with the data (CRUD operations), and the additional code and code complexity would mean a longer implementation time. Also considering the previously decided architectural decisions, it could potentially clash with those.
- Layered Architectural Pattern
  - The Layered architecture could be a good fit since the data caching and sharing are not integral to the core service of storing data, but rather are additions to enhance the core service. They are layered on top.
- Repository Architectural Pattern
  - Seems like a good fit due to the strong connections to database features. Data is being stored in and modified through the main database (acting as the repository). This architecture also lends itself well to long-term data storage, which is a key selling point.
- Pipe and Filter Architectural Pattern
  - Similarly to the layered pattern, pipe and filter could be a good fit. The database operates on receiving input through requests from the user, and appropriately provides output based on the input. The input from the user, although not directly a “transform”, does end up with data within the database being transformed in some cases. All “transforms” are sequential, with no parallelism.
- Creational Design Pattern
  - Prototype

        - Can use a prototype for both private and public database entries and instantiate the prototype
- Structural Design Pattern
    - Flyweight
        - Multiple object instances (database entries) that need to be supported at once
- Behavioral Design Patterns
    - Template Method
    - The template method seems like a good fit for the project since we can define the action of creating a new database entry as a skeleton of an algorithm. The main step that changes depending on entered information is where to store the information depending on if the user selected it to be public or private. This operation can be deferred to a subclass, where it can be defined separately for public or private data entry.
- Concurrency Design Patterns
    - The Thread Pool design pattern can be used to increase performance in the application
    - By queuing up tasks, the database can be updated quicker and more often, potentially even leading to real-time updates
    - Threads can be set to perform specific tasks (i.e. creating new entries, editing entries, deleting entries, fetching entries, etc.) that way concurrent operations in the database can occur
    - Main issue would be race conditions if a thread attempts to change an entry before it is finished being created

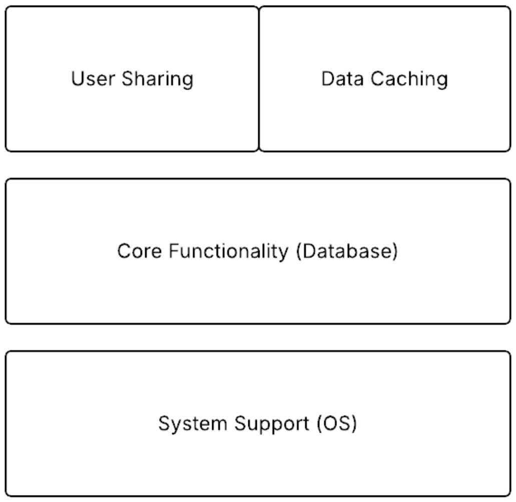
Selected Architectural Designs: Layered Architecture, Prototype Creational Architecture, Flyweight Structural Architecture, Template Behavioral Architecture

Step 5: Instantiate architectural elements, allocate responsibilities and define interfaces

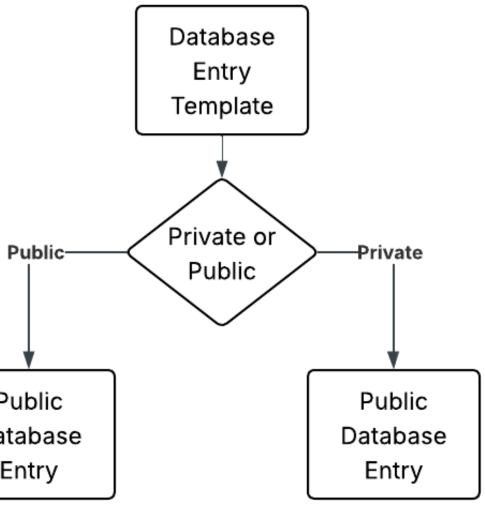
Design Decision and Location	**Rationale**
Implement data caching in the database	In line with the Layered Architectural design, data caching can be implemented on top of the core features to allow for offline access when an internet connection is unavailable.
Implement catch sharing between users	In line with the Layered Architectural design, catch sharing can be implemented on top of the core features to allow users to make their catches private or public. Public catches can be viewed by other users who are utilizing the service.
Adopt more architectural designs that fit the project	Additional architectural designs that fit the project will be adopted. Namely, the Prototype Creational Design Pattern, Flyweight Structural Design Pattern, and the Template Behavioral Design Pattern will be adopted. These design patterns fit within the design purpose of the project and will help to better focus the scope and development efforts of the project.

Step 6. Sketch Views and Record Design Decisions

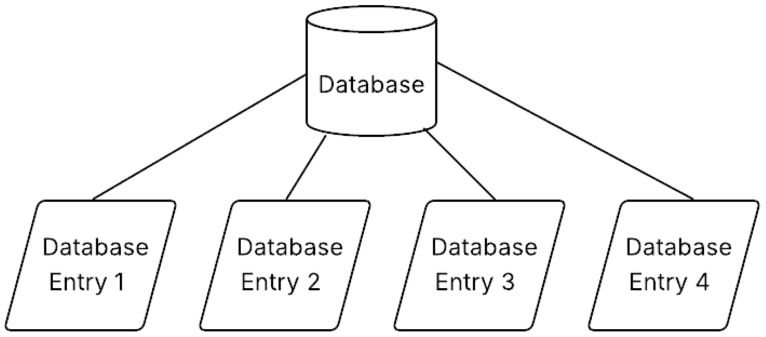
Layered Architecture Diagram



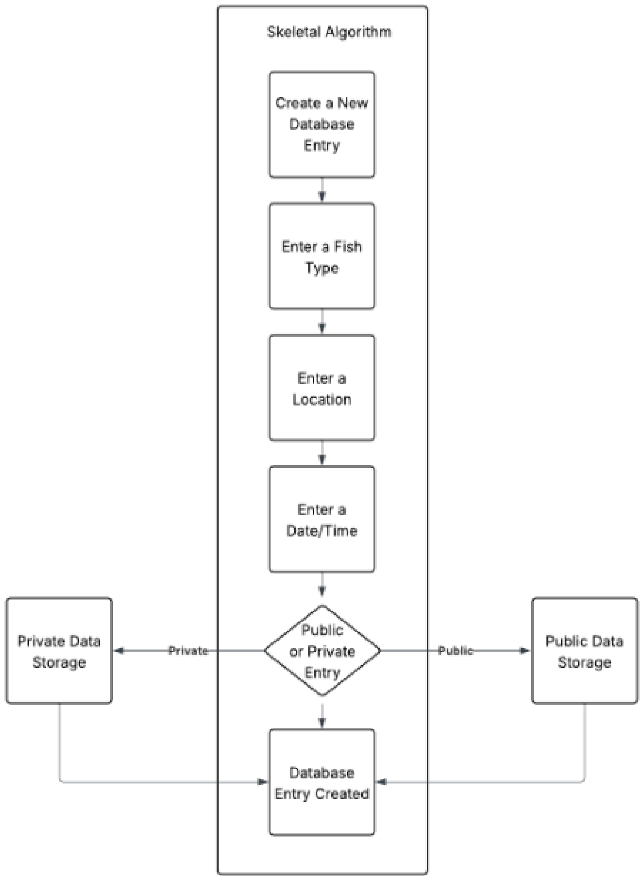
Creational Design: Prototype Diagram



Structural Design: Flyweight Diagram



Behavioral Design: Template Diagram



Element	Responsibility
Data Caching Layer	The data caching layer will ensure that a user’s own database entries will remain available if the internet is unavailable. Given that some fishing spots are in more remote regions, this allows fishermen to access their data while out on a trip. (QA-3, CON-2, CRN-2)
Data Sharing Layer	The data sharing layer will allow users to elect to share their catches with other users on the service. They can choose whether or not to include information such as their real name when sharing. (QA-1, QA-2, CON-3)

Step 7. Perform Analysis of Current Design and Review Iteration Goal and Achievement of Design Purpose

Not Addressed	Partially Addressed	Completely Addressed	Design Decisions Made During the Iteration
		UC-1	Used the Fat Two-Tier Client-Architecture to allow users to connect to the database and store/view catch information. Offline access in the form of data caching.
		UC-2	Textual Data Storage allows pro-fishermen to view important catch information to analyze and extrapolate data to better make decisions. Offline access in the form of data caching.
		UC-3	Textual and Image Storage allows more general wildlife entries to be made. Only manual entry is available for this feature. Offline access in the form of data caching.
		UC-4	Users can share their catch information with other users, with the option to turn them back to private at any time. Users can also choose to share some of their personal information such as their

			name, but will never be forced to.
		QA-1	Users can share their catch information with other users.
		QA-2	Any personal information entered by the user will be stored securely on the server. Security features mentioned previously will keep information safe.
		QA-3	Offline caching now is fully implemented, allowing users to access certain database information while an internet connection is unavailable.
		QA-4	Due to the selected architecture, processing load is dependent on the client and not the server, therefore processing load should be well distributed geographically.
		QA-5	Due to the selected architecture, the database itself should not have much processing load relegated to it, therefore it will be speedy in its updates.
		QA-6	Text and Image Storage allows for long term data storage in the database.
		QA-7	Once the service enters production, a separate branch will be utilized for the implementation of new features.
		CON-1	Some of the newly adopted architecture designs work well when handling large volumes of long-term data.
		CON-2	Now that data caching is implemented, offline access is available for certain database information.
		CON-3	Now that the data sharing layer has been implemented, shared data can be stored separately from private data.

		CRN-1	Users can create, read, update, and delete their own database entries.
		CRN-2	Now that data caching is implemented, offline access is available for certain database information.
		CRN-3	The adoption of the Fat Two-Tier Client-Server architecture means that new features and the expansion of current features can be implemented easier.