Cloud Pet Design Doc

Table of Contents

ℹ️ Overview

Cloud Pet is like having a digital pet that lives on your phone or computer. You can play with it, feed it, and clean it. Because it's built using cloud technology, it's always available and can handle multiple users caring for their pets.

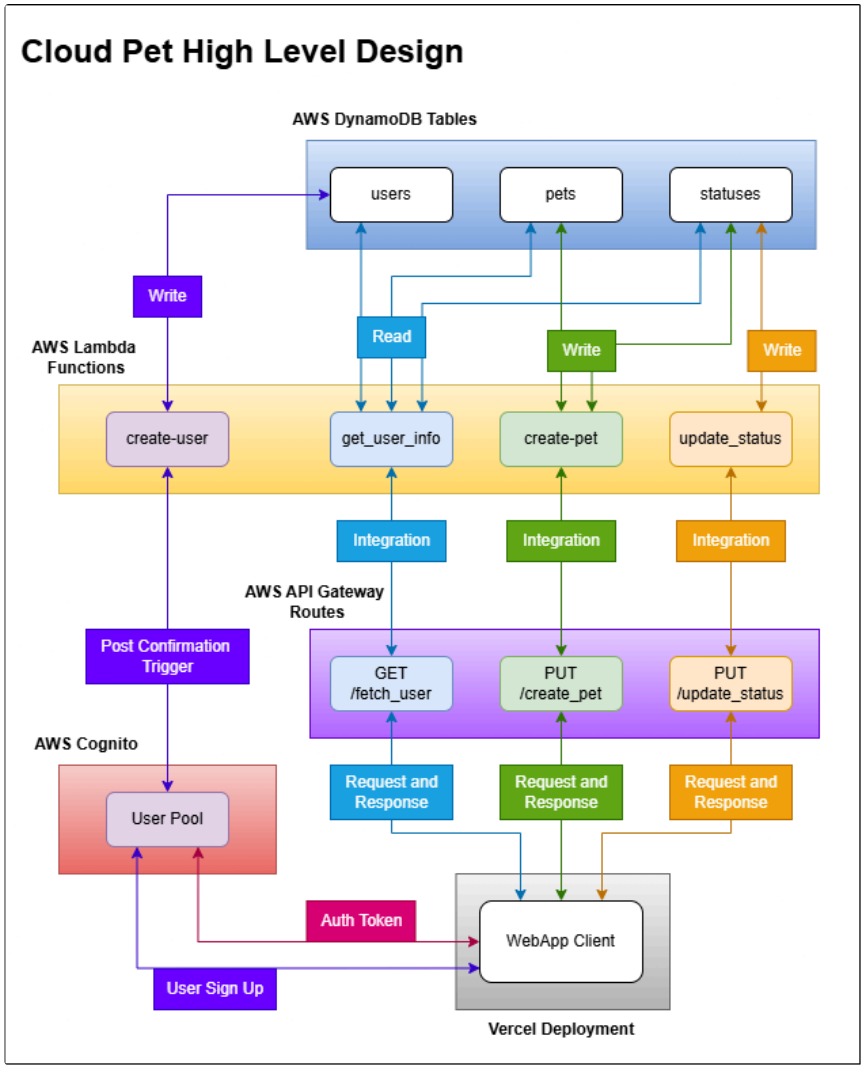
⚙️ Functional Requirements

1. User Registration & Authentication
   a. Users can sign up, log in, and manage their pet.
   b. Each user can have one or multiple pets.
2. Pet Interaction System
   a. Users can feed, clean, and rest their pets.
   b. Users can view pet stats like overall happiness, hunger, sleep, and hygiene that change over time.
3. ~~Push Notifications & Alerts~~
   a. ~~Users can opt into notifications for pet hunger or mood changes~~

🧩 Non-functional Requirements

1. Performance & Scalability: The service should handle multiple users and pets concurrently.
2. Security: The service should prevent users from accessing data that is not their own.
3. Availability & Reliability: Pet state should persist even if the user logs out.
4. Maintainability: The service should be designed for easy feature updates and extension.

📊 High Level Design



☁️ Cloud Technologies and Services

Cloud Technology	Use Case	Design Choice Rationale
Vercel	Vercel is used to deploy our Next.js client WebApp. It deploys from a GitHub repository with CI/CD pipelines.	Vercel was chosen over other deployment options, such as AWS Amplify, because of its simplicity and quick start-up. Deploying our WebApp client on Vercel helps us quickly develop and launch a frontend in Next.js (with built-in GitHub integrations).
AWS API Gateway	AWS API Gateway is used to form and handle routes on the backend. Routes are integrated with AWS Lambda to interact with the AWS DynamoDB.	AWS API Gateway is a natural choice to integrate with AWS Lambda functions, allowing for a simple route (with a separate domain) set up within the AWS ecosystem.

AWS Lambda	AWS Lambda is used for CRUD functionality on the DynamoDB tables – whether through AWS API Gateway routes or by AWS Cognito Triggers.	AWS Lambda was chosen for its easy integration with other AWS services like DynamoDB and Cognito. Additionally, Lambda encapsulates and obscures back-end functionality from the WebApp.
AWS DynamoDB	AWS DynamoDB is our data storage solution, using tables that are interacted with by AWS Lambda functions.	AWS DynamoDB was chosen over other databases like RDS for its alignment with our project's serverless approach (AWS Lambda) and simplicity. Additionally, this project lacks the need for complex queries that make SQL preferable.
AWS Cognito	AWS Cognito enables user account functionality, including logins, signups, and authentication/authorization.	AWS Cognito was chosen over other options like Google OAuth for easy integration with AWS services like Lambda, and its simple application on the front-end.

⚠️ Limitations

Overall this service is designed to be modular and easily extensible. However, this does not mean there are no limitations on the current design and implementation. Currently the biggest limitation we face is enabling diverse status interactions (due to time-constraints).

For example, statuses are generic to the type they represent (sleep, hunger, etc.). While this is great for extensibility, there are currently no systems in place to allow unique interactions for statuses, particularly sleep. Ideally, sleep would be interacted with over-time (increasing over-time as the pet sleeps), however, the current generic statuses only allow for one-time increments in value.

Working around this limitation is possible, but would be the first problem addressed in future iterations of this project. Diverse status interactions would enable more ways for the user to interact with their pet (not just for sleeping/resting).

🏛️ Architecture

📪 API Routes

PUT Route `/create_pet`

Integrated with the create-pet Lambda function. See the definition for this function for more details.

Request

```
{
  "method": "PUT",
  "url": "<ENDPOINT>/create_pet",
  "headers": {
    "Authorization": "Bearer <AUTH_TOKEN>",
    "Content-Type": "application/json"
  },
  "body": "{\"userId\":\"<USERID>\",\"name\":\"<PET_NAME>\"}"
}
```

Success Response (200 OK)

```
{
  "statusCode": 200,
  "body": {
```

```
    "petId": "<PETID>",
    "statusIds": [
      "<STATUSID_1>",
      "<STATUSID_2>",
      "<STATUSID_3>"
    ]
  }
}
```

Bad Request (400 Bad Request)

```
{
  "statusCode": 400,
  "body": {
    "message": "Missing userId or name"
  }
}
```

Internal Server Error (500 Internal Server Error)

```
{
  "statusCode": 500,
  "body": {
    "message": "Internal Server Error",
    "error": "Some detailed error message from AWS SDK"
  }
}
```

PUT Route `/update_status`

Integrated with the update_status Lambda function. See the definition for this function for more details.

Request

```
{
  "method": "PUT",
  "url": "<ENDPOINT>/update_status",
  "headers": {
    "Authorization": "Bearer <AUTH_TOKEN>",
    "Content-Type": "application/json"
  },
  "body": "{\"statusId\":\"<STATUSID>\",\"petId\":\"<PETID>\", \"incrementValue\":\"<INCREMENT_VALUE\"}"
}
```

Success Response (200 OK)

```
{
  "statusCode": 200,
  "body": "Updated <EXAMPLE_STATUS_ID> for pet <EXAMPLE_PET_ID> to <EXAMPLE_NEW_VALUE> at <EXAMPLE_TIMESTAMP>"
}
```

Bad Request (400 Bad Request)

```
{
  "statusCode": 400,
  "body": "Missing parameters"
}
```

Not Found (404 Not Found)

```
{
  "statusCode": 404,
  "body": "Status not found"
}
```

Internal Server Error (500 Internal Server Error)

```
{
  "statusCode": 500,
  "body": "Internal server error"
}
```

GET Route `/fetch_user`

Integrated with the get_user_info Lambda function. See the definition for this function for more details.

Request

```
{
  "method": "GET",
  "url": "<ENDPOINT>/fetch_user?userId=<USER_ID>",
  "headers": {
    "Authorization": "Bearer <AUTH_TOKEN>",
    "Content-Type": "application/json"
  },
  "body": null
}
```

Success Response (200 OK)

Returns all pets each with their three statuses (simplified view here).

```
{
  "statusCode": 200,
  "body": {
    "user": {
      "userId": { "S": "<EXAMPLE_USER_ID>" },
      "name": { "S": "<EXAMPLE_USER_NAME>" },
      "email": { "S": "<EXAMPLE_USER_EMAIL>" }
    },
    "pets": [
      {
        "petName": "<EXAMPLE_PET_NAME>",
        "petId": "<EXAMPLE_PET_ID>",
        "statuses": [
          {
            "statusId": "<EXAMPLE_STATUS_ID>",
            "petId": "<EXAMPLE_PET_ID>",
            "type": "<EXAMPLE_STATUS_TYPE>",
            "decrementRate": "<EXAMPLE_DECREMENT_RATE>",
            "incrementValue": "<EXAMPLE_INCREMENT_VALUE>",
            "lastTimestamp": "<EXAMPLE_TIMESTAMP>",
            "lastValue": "<EXAMPLE_LAST_VALUE>"
          }
        ]
      }
    ]
```

```
  }
}

```

Bad Request (400 Bad Request)

```
{
  "statusCode": 400,
  "body": {
    "error": "Missing userId"
  }
}
```

User not Found (404 Not Found)

```
{
  "statusCode": 404,
  "body": {
    "error": "User not found"
  }
}
```

Pet not Found (404 Not Found)

```
{
  "statusCode": 404,
  "body": {
    "error": "No pets found"
  }
}
```

Internal Server Error (500 Internal Server Error)

```
{
  "statusCode": 500,
  "body": {
    "error": "<EXAMPLE_ERROR_MESSAGE>"
  }
}
```

λ Lambda

Function `create-user`

An AWS Cognito Post Confirmation Trigger invokes this function. It creates a single row in the users table using the `sub` from the User in Cognito.

Only the AWS Cognito can invoke this function.

Parameter	Definition
userId	The `userAttributes.sub` from AWS Cognito.
username	The `userAttributes.preferred_username` from AWS Cognito.
email	The `userAttributes.email` from AWS Cognito.

DynamoDB Table	Changes
users	Creates a single row using the passed parameters, `createdAt` is assigned the value of the current DateTime.

Function `create-pet`

This function creates a pet (and its status rows) with a given name for the passed userId. It takes two parameters, `userId` and `name`, modifying the `pets` and `statuses` tables in DynamoDB.

Only a user belonging to `userId` may create pets under `userId`.

Parameter	Definition
userId	ID of the parent user.
name	The name for the pet.

DynamoDB Table	Changes
pets	Creates a single pet row with the passed `name` under `userId`. Creates a random UUID for `petId`. Values for statusIDs are random UUID associated with each status.
statuses	Creates three status rows of type hunger, sleep, and hygiene under the newly created `petId`. Creates a random UUID for `statusId`. Default `lastValue` is 100, `lastTimestamp` is now(), and `decrementValue` is 0.5 (per minute).

Function `get_user_info`

This function retrieves basic user info along with an array of the user's pets (including pet status) given the `userId` . Upon retrieval from the tables, the function calculates and updates each pet's status values based on the last timestamp in which the status was updated. In this way, pet statuses decrease over time.

Parameter	Definition
userId	ID of the user (obtained from session)

DynamoDB Table	Changes
statuses	For each row containing the `pet_id` of pets belonging to the current user the new `lastValue` is calculated using the `decrement_value` and `lastTimestamp`. `lastValue` and `lastTimestamp` are then updated accordingly.

Function `update_status`

This function updates the status bar of a pet by the given increment. It takes the following parameters: `status_id` (Ex: 123, 345), `pet_id` (Ex: pet123), and the `increment_value` (Ex: 10), modifying the `statuses` table in DynamoDB.

Parameter	Definition

statusId	ID of the status.
petId	ID of pet.
incrementValue	value to increase the status bar by

DynamoDB Table	Changes
statuses	Updates the row with the given `status_id` and `pet_id` and increments the `lastValue` by the `increment_value` and updates the `lastTimestamp` with the current time

💾 DynamoDB

This architecture will employ three DynamoDB tables as the database, using `users`, `pets`, and `statuses`.

Table relationships are established by including their parent's id as a sort key/secondary index. For example, each row in the `pets` table will have `userId` as a sort key, and each row in the `statuses` table will have `petId` as a sort key.

Table `users`

The `users` table contains basic information about their user with an id associated with the AWS Cognito `User pool - Cloud Pet`.

Rows on the `users` table are only created by a Post Confirmation Trigger by AWS Cognito (after successful user registration).

userId	email	username	createdAt
The `userAttributes.sub` value from AWS Cognito.	The email used by the user for registration.	The username used by the user for registration.	The timestamp taken during user registration.

Table `pets`

The `pets` table contains pet information and has a sort key for their parent `userId`. Additionally, each row contains a `hungerStatusId`, `sleepStatusId`, and `hygieneStatusId` which relate to rows in the statuses table.

Rows on the `pets` table are created by the same lambda function as statuses. When the API route to create a pet is invoked, it will first create the statuses, then the pet (assigning the status UUIDs).

petId	userId (sort key)	name	hungerStatusId	sleepStatusId	hygieneStatusId
UUID.	ID of user (“owner”).	The pet’s name.	UUID of status for hunger.	UUID of status for sleep.	UUID of status for hygiene.

Table `statuses`

The `statuses` table contains information for a single status and has a sort key for its parent `petId`. Additionally, each row contains a `lastTimestamp` which is the timestamp for the `lastValue`.

currentValue = lastValue - [(currentTimestamp - lastTimestamp) x decrementRate]

statusId	petId (sort key)	type	lastValue	lastTimestamp	decrementRate

UUID.	UUID of pet.	The status type (i.e. hunger)	The last recorded value of status.	The timestamp taken at lastValue.	The rate the status decreases each minute

🔐 Authentication and Authorization

🪪 AWS Cognito Login

Login functionality with user authentication is implemented using AWS Cognito.

Using React hooks, the frontend page checks if there is a user session. If not, it prompts the user to sign in. This redirects to the AWS Cognito page tied to the `User pool - CloudPet` (a secure storage for user login information). On this Cognito page, the user can sign in or sign up.

Upon a successful sign-up (i.e. new user), the Cognito invokes a Post Confirmation Trigger for the create-user Lambda function, which writes the user to the users table in the DynamoDB.

📧 Auth Tokens on Requests

Authentication tokens and sessions are managed using NextAuth.js with AWS Cognito as the authentication provider.

Session Structure

```
async session({ session, token }) {
      session.user.id = token.id as string;             // Cognito sub
      session.user.username = token.username as string; // Cognito username
      session.accessToken = token.accessToken as string | undefined;
      return session;
    }
```

The `accessToken` is obtained from Cognito during authentication and stored in the session. This way, the token and `userId` can be easily accessed by the frontend and used to make requests to the API.

In the future, we can store the accessToken in an encrypted JSON Web Token (JWT) for additional security, which is also supported by NextAuth.js

🐶 Pet State Management

The Pet State Management system is responsible for tracking and updating a pet's core attributes, ensuring an interactive and engaging user experience. This feature includes key metrics such as hunger, sleep, hygiene, type, name, and status, which dynamically change based on time and user interactions.

- Hunger decreases over time, requiring players to feed their pet.
- Sleep determines the pet's energy levels, affecting its activity.
- Hygiene tracks cleanliness, prompting players to groom their pet.
- Type represents different pet species, each with unique traits.
- Name allows for personalization and attachment.
- Status provides an overall summary of the pet's condition and mood.