Cloud Computing Project Report

Authors:

Daniel Martins (ID 73951)

João Ribeiro (ID 73706)

Introduction:

This project aims to build the backend system for a web application that manages collections of Lego sets and allows users to create and participate in Lego auctions. Each user can register, upload photos for their profile, add their Lego sets, leave comments, and sell their own Lego sets through auctions.

The backend is implemented as a REST API and stores all generated data using Cosmos DB and Azure Blob Storage. User profile pictures and Lego set images are saved in Blob Storage, while session data and structured information are stored in Cosmos DB.

The application is hosted on Azure App Service, and Redis Cache is used to improve performance. The system was tested locally using Payara and endpoints were manually verified using Insomnia.

System Design:

The system is designed to allow users to manage personal Lego collections and participate in auctions. Users can create and update their profiles, including email and profile pictures.

Administrators are responsible for adding new Lego sets, providing detailed descriptions, and uploading related images. Users can also leave comments on specific Lego sets, fostering interaction and community engagement.

Auctions allow users to sell their Lego sets and bid on sets from other users. Each auction references the user who created it and the associated Lego set, and includes information about the starting bid and closing time.

Data Model (Cosmos DB)

Container Name:	Stores:
Auctions	id - id of the auction startingBid - the starting price of the auction bid auctioneer - id of the user who created the auction Product - id of the manufactured legoset createdAt - timestamp of when the auction was created endsIn - timestamp of when the auction is to be closed currentBid - the highest current bid value of the auction closed - a boolean value to know whether the auction is still ongoing or not
Comments	id - id of the comment

	productId - id of the legoset author - id of the user that wrote the comment Content - String that store the comment of the user Timestamp - timestamp that stores when the comment was created
LegoSet	id - id of the legoset name - String that stores the name of the legoset description - String that stores description of the legoset photos - id of the blob storage that has saved the image of the legoset yearOfProduction - year of the date the legoset was created
Manufactured	id - id of the manufactured legoSetId - id of the legoset its associated with owner - id of the user that owns this copy of the legoset createdAt - timestamp of when the row was created
Media	id - id of the media name - original name of the uploaded file file - name of the file in the Images Blob container mediaType - tag that stores what type a file is uploaded owner - id of the user that created this row createdAt - timestamp of when the row was created
Sessions	id - id of the session user - id of the user that created the session profile - stored used data that is stored in the cache createdAt -timestamp of when the row was created
Users	id - id of the user Username - String that stores username of user dataOfCreation - timestamp of when the user was created in the database Avatar - media Id of his uploaded profile image Power - stored value that is used to determine the admin power of users in the webapp Email - String that stores the email of the user Password - String that stored the hashed password of the user lastUpdate - timestamp of when updated his account isDeleted - a Boolean that stores whether the user has been deleted or not

User API Endpoints

EndPoint :	Method:	Description:
/auth/login	POST	Authenticates a user using their email and password. If successful, it creates a session token, stores it in Redis and Cosmos DB, sets a **Session** cookie in the response, and returns the user profile. Returns **400** if email or password is missing, and **401** if authentication fails.
/auth/logout	GET	Logs out the currently logged-in user by deleting their session from Redis and Cosmos DB. Also invalidates the **Session** cookie in the client by setting it to null and expiring it. Returns a confirmation message "**Disconnected**".

/auth/register	POST	Registers a new user with username, email, and password. If successful, it creates a new session, stores it in Redis and Cosmos DB, sets a **Session** cookie, and returns the newly created user profile. Returns **400** if required fields are missing and returns **Null** if the email is already in use.
/auction/create	POST	Creates a new auction for a legoset. The user must be logged in (session checked via cookie) and must own the product. Validates that the auction data contains a product and a timestamp with the closing date(that endsIn is ≤ 60 minutes) and that no existing auction exists for that product. Returns the created auction on success (201 Created), or appropriate error statuses for invalid requests or unauthorized access.
/auction/bid/{id}	POST	Place a bid on an existing auction. The user must be logged in. Checks if the auction exists and is still open. Ensures the bid is higher than the starting bid and the current highest bid. Updates the auction with the new bid if valid, and returns a status message (Success or an explanation if the bid is invalid).
/auction/list	GET	
/auction/view/{id}	GET	Retrieves a specific auction by its ID. Returns detailed information including the auction ID, starting bid, seller profile, legoset product details, creation time, and duration (endsIn). Returns null if the auction does not exist.
/media/upload	POST	Uploads an image file to Azure Blob Storage. The user must be authenticated via a Session cookie. Rejects files larger than 128 KB. If the upload succeeds, returns 201 Created; otherwise, returns appropriate error codes such as 400 Bad Requests (unauthenticated), 509 Bandwidth Limit Exceeded (file too large), or 500 Internal Server Error (upload failed).
/media/download /{id}	GET	Downloads a media file by its id in the cosmos DB. Retrieves the file from Azure Blob Storage and returns it with the correct content type. Returns 200 OK with the file stream if found, or **null** (implicitly 404 Not Found) if the file doesn't exist.
/legoset/{id} /comment/post	POST	Authenticated users can post a comment on a specific Lego set. Returns the created Comment object. Returns null if the user session is invalid.
/legoset/{id} /comment/list	GET	Retrieves a list of comments for a given Lego set. Supports an optional before query parameter to filter comments created before a specific timestamp.
/legoset/create	POST	Creates a new Lego set. Requires the user to be authenticated and have admin privileges (power ≥ 3). Returns the created LegoSet. Returns 401 Unauthorized if the user is not authenticated or lacks privileges.
/legoset /manufactured /create	POST	Creates a new manufactured copy of a Lego set and optionally assigns it to an owner. Requires admin privileges. Returns the created ManufacturedData. Returns 406 Not Acceptable if the Lego set ID is missing.
/legoset /manufactured /assign/{id}	PUT	Updates the ownership of an existing manufactured legoset. Requires **admin privileges**. Returns the updated **ManufacturedData**. Returns **406** Not Acceptable if the manufactured Id is invalid or not found.
/users/retrieve	GET	Retrieves a specific user profile by user ID. Returns the **UserProfile** object that stores all the user data in a secure way.
/users/list	GET	Returns a paginated list of **user profiles**. Supports an optional **token** parameter for pagination(Returns a map containing profiles and metadata).

/users/profile	GET	Retrieves the profile of the currently logged-in user using the session cookie. (Returns the UserProfile)
/users/update	PUT	Updates the profile of the currently logged-in user. Accepts a JSON body with updated fields and requires the session cookie. Returns the updated UserProfile (Returns null if the session is invalid).
/users/delete	DELETE	Deletes the currently logged-in user. Requires a valid session cookie. Logs out the user after deletion. Returns HTTP 200 with a success message, or 403 if the user is not logged in.

Evaluation:

Testing Approach

Initially, the plan was to evaluate the system using **Artillery** to perform automated performance and load testing across different deployment configurations.
However, due to configuration and connectivity issues, we were **unable to properly execute automated tests** using Artillery.

Instead, we performed a **structured manual testing process using Insomnia**, where each endpoint was individually tested.
This included all **GET**, **POST**, **PUT**, and **DELETE** operations across the main controllers (User, Media, LegoSet, Comment, and Auction).
The testing was conducted in a controlled environment, with **hardcoded request data** to simulate real-world API interactions such as user registration, media upload, auction creation, and bidding.

Observations

- All endpoints responded as expected under manual testing, returning valid HTTP statuses and appropriate JSON responses.
- Authentication cookies were successfully handled by Insomnia during testing of protected endpoints.
- Media uploads and downloads worked correctly, validating Blob Storage integration.
- Redis caching behaved correctly — cached data was returned for repeated user lookups, and invalidated upon updates.

While this testing method does not provide automated performance metrics, it allowed for thorough functional validation of each endpoint and ensured that the system behaved correctly across its core features.

Conclusions

This project demonstrates how cloud computing services can be combined to build a scalable and reliable backend for a Lego collection and auction system.

The use of **Cosmos DB** for storage, **Blob Storage** for media management, **Redis** for caching, and **Azure Functions** for automation provided a robust and efficient architecture.
Manual testing via Insomnia confirmed that all core functionalities operate correctly and integrate seamlessly with Azure resources.

Future improvements could include adding automated testing (using Artillery or similar tools), introducing search capabilities, or implementing analytics and recommendations using Azure Cognitive Search or Spark.