TO DO App documentation

Szymon Rećko, Hubert Ruczyński, Mateusz Sperkowski

Table of contents

Project description

TO DO App is an online website hosted on the AWS servers that enables the user to create and log in to their own accounts, and create their workspace with easy to manage TO DO list. Additionally, the user can easily filter the tasks by their completion state or the due date. The project is deployed behind this link: https://m073n8ce94.execute-api.us-east-1.amazonaws.com/stage/register.

Functional requirements

Name	Description
Retrieval of accomplished tasks	Allow users to retrieve all the tasks in their to-do list.
Customizable tasks creation	Allow users to create a new task in their to-do list with a title, description, status, and due date.
Tasks updates	Allow users to update an existing task's title, description, status, and due date.
Monitoring tasks status	Allow users to mark a task as TO DO, in progress, or completed, and remove it from the to-do list.
Tasks filtering	Provide an option for users to filter tasks by their status (in progress, completed, or incomplete) and due date.
Registration and login*	Allow users to create their personal accounts and safely log in to the system.

Table 1 The description of the functional requirements of the project.
** Denotes an additionally added functionality.*

Non-functional requirements

Name	Description	Requirement Assurance
Scalability	Handle an increase in traffic without any performance degradation.	1. Scalable DB instances from 100GiB to 1000GiB. 2. The Lambda scales up to 400 concurrent workers.
Reliability	Provide a consistent level of service even in the face of hardware or software failures.	1. Multi-AZ deployment with a Database standby copy in different AZs 2. The Lambda is available in 2 AZs which makes it resilient to failovers.
Security	Protect against unauthorized access and data breaches.	1. Database secured with login and password, available only from instances in the same VPC, and deployed in the private subnet. 2. Hashed account passwords.
Performance	Low response times available.	1. Extremely lightweight serverless implementation with API calls. 2. Only aggregated read/write operations to and from the database.
Cost optimization	Minimize infrastructure costs while still meeting the application's needs.	1. The use of cheap db.t3.micro instances. 2. Lack of extensional expenses (ex. 2nd Region). 3. Lack of redundant features (ex. RDS proxy).

Table 2 The description of the non-functional requirements of the project.

Cloud architecture

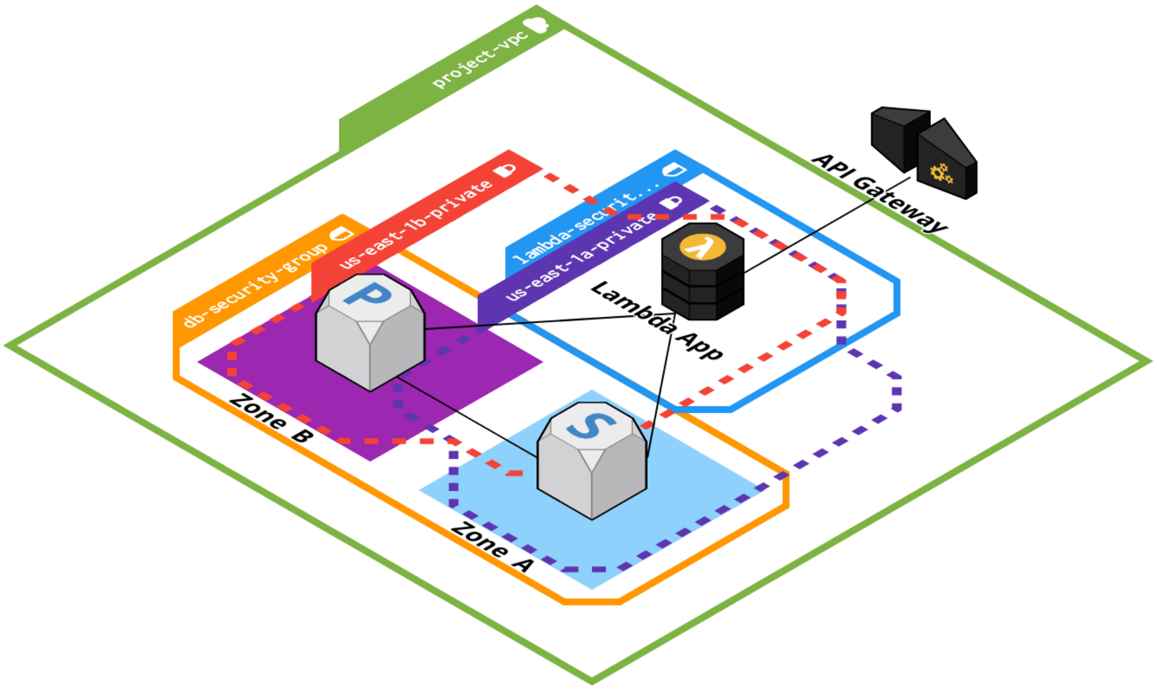


Figure 1 Cloud architecture diagram describing the TO DO application deployment in AWS, generated with the Cloudcraft website.

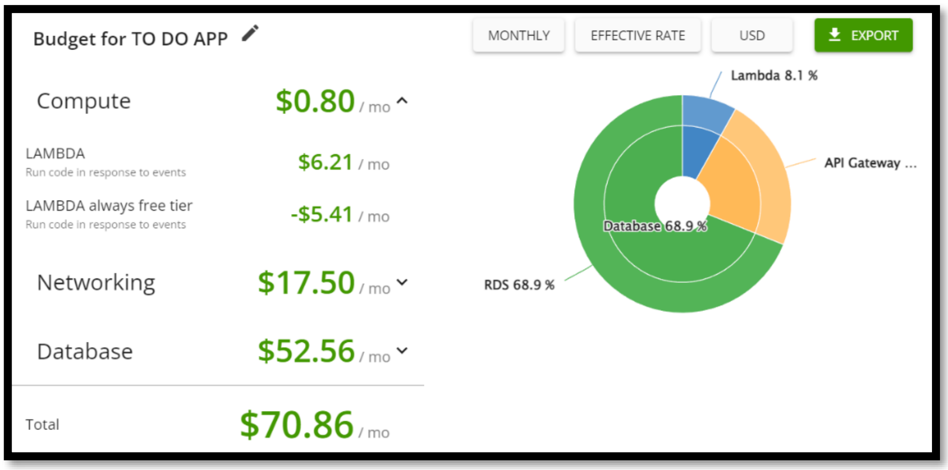
Description

The diagram presented in Figure 1 represents the deployment of our application inside AWS. The main services used here are the API Gateway, AWS Lambda, and two Amazon RDS PostgreSQL databases. The API Gateway is connected to the AWS Lambda instances, and it is the only object outside the project-vpc network, which is the only VPC in this project. The second, and most important service are the AWS Lambda instances, connected to the gateway and both database instances. This object has an individual security group assigned, to ensure that only qualified developers will be able to modify its performance in the future. Additionally, Lambda instances are deployed across two Availability Zones A and B, which ensures that even if something happens to one of them, the other will be able to handle the traffic properly. This mechanism makes sense, because of two PostgreSQL instances that are also deployed in these Zones, where the main database resides in Zone B, and a standby copy is inside Zone A. Both databases are assigned to the same security group, as their maintainers require access to both.

The whole application is deployed within the US-East-1 (Northern Virginia) region as enabling the second region (us-west-1, Oregon) would duplicate our costs which is contradictory to the cost optimization non-functional requirement. However, we are aware that such a solution would increase the prevalence of our data and increase the availability of the app. Another alternative solution would be adding the third AZ with another standby copy of the database however, as our current infrastructure relies on the PostgreSQL instances, adding another one is costly, as it generates big costs, presented in the next section. We could also reorganize our data storage into non-relational databases, which would be around two times cheaper for our expected traffic (described in the next section) and this way provide another AZ or even the region within the same budget. For example, NoSQL data storage costs only appear during requests to the database, while in our version there are constant costs of upkeep.

Costs

With the usage of the Cloudcraft website, we were also able to estimate the costs of the presented architecture. As plenty of used services costs depend on the network traffic, we estimate the costs for 5 million queries per month (which would be enough for 3000 users, making 50 calls per day for 30 days). As mentioned in the previous section, and visible in Figure 2, the main costs of our application come from the PostgreSQL databases, and another major part of the costs is the API Gateway networking infrastructure. Another worth mentioning fact is that the real computational costs of AWS Lambda equals 6.21$/month however, according to Cloudcraft 5.41$/month is discarded as a free tier.



Budget for TO DO APP

Compute	$0.80 / mo
LAMBDA Run code in response to events	$6.21 / mo
LAMBDA always free tier Run code in response to events	-$5.41 / mo
Networking	$17.50 / mo
Database	$52.56 / mo
Total	$70.86 / mo

Figure 2 Estimated costs for AWS architecture for one month and 5 million queries per month.

Technological stack

In this section, we describe the technologies used in the development of our application, which consider both the strict app implementation and the AWS services configuration.

Application deployment

App Implementation

Our application is implemented as an AWS Lambda function written in Python 3 with frontend done in HTML and Javascript. To make it easier for users, cookies are stored in the browser, which means that after logging in once users will have fast access to their data.

AWS Lambda

The backend is done in the FastAPI framework (MIT License) that connects to the database using the psycopg2 library (GNU Lesser General Public License). The application has to be adapted to Lambda using the Mangum library (MIT License). To ensure security, passwords are hashed using the des_crypt algorithm from the passlib library (BSD license). However, due to the innate Lambda scaling of workers, many database connections are opened as existing workers. Since our database is set to a maximum of 100 concurrent connections, this limits the number of workers to 100. One possible solution is to extend the number of connections or use the pricey RDBS proxy. This service would allow us to use the same connections between different workers, so we would no longer open one connection per each and no longer restrict the worker count. Initially, we set up this option, however, due to high costs and our limited budget, we had to

disable this option.
Vulnerable parameters, such as database connection information, are stored as Lambda Parameters outside of the code. This means that even if the code to the application leaks, there are no security risks beyond that.

API Gateway

API Gateway is the key service that allows for invoking our app API for our users. It deals with all the concurrent calls, their responses and networking, and authorization. It serves the purpose of a front door to the application. It has no running fees, only per-request ones, which fits perfectly into our architecture and limits the working costs.

Data storage

As the data storage engine, we've decided to use a relational PostgreSQL as it seemed like a very simple and effective solution. Additionally, we were aware that the Amazon RDS enables a really simple Multi-AZ deployment thus we've decided to choose it as our backbone. After implementing the app, we've understood that for the application with a small user base, it would be better if we used the non-relational approach, as it is billed per request only and does not include any constant costs. This way we would be able to reduce the costs or provide a third AZ or even deploy the application in the second region.
Additionally, during this part of the implementation, we've learned that additional standby instances, instances in the new region, and Amazon RDS Proxy are billed equally per instance, which resulted in consuming 13$ in two days. In comparison, another 2 weeks were consumed 50$ only, even though the app was fully functional from this point.

AWS RDS: PostgreSQL

Structure

As presented in Figure 3, the database is very simple and consists of only two tables corresponding to the login and registration process, and the second one stores information about particular tasks.
The login table keeps the information about the username, and encrypted password, and yields the personal token assigned to the account, which determines the cookie sent to the user after logging in.
The tasks table keeps information about the tasks for a particular user. We've used a primary key combined of the tuple task_id and token, which ensures that the tasks are assigned to the correct user, and can be enumerated for each of the users separately. Other fields describing the tasks are their Name, Description, State (encoded in the database as 1, 2, or 3), and its Due_date.

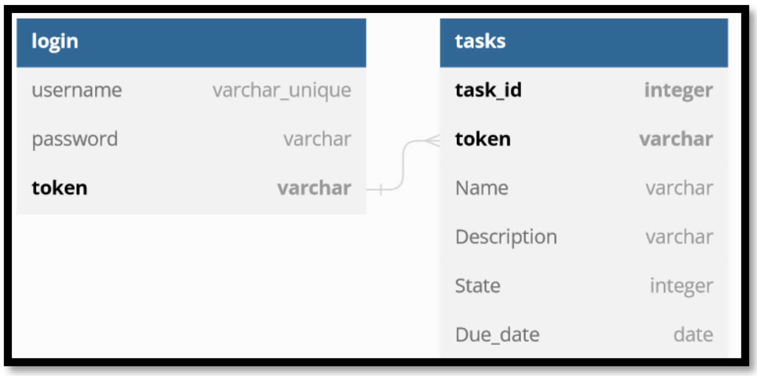


Figure 3 PostgreSQL Database structure.

Configuration

Parameter	Value	Comment
Engine	PostgreSQL 14.6-R1	
Deployment type	Multi-AZ DB Instance	For data security and higher availability.
Instance type	Burstable db.t3.micro (2v CPIs, 1GiB RAM, 2085 Mbps)	The smallest available solution, as the app won't have insane traffic. The t4.micro type was not enough.
Storage	Allocated storage: 100GiB Autoscaling to 1000GiB Provisioned IOPS: 1000	The stored information is small, thus we've used the smallest amount of resources possible, but still ensuring the basic scalability during the maintenance period. We are aware that in case of a huge burst, it might be not enough, and other solutions might be needed.
Connectivity	VPC: project-vac Public Access: No Security group: db-security-group Database port: 5432 RDS proxy: No	We ensure that the database is within the project's VPC in order to make it publicly inaccessible, which results in enhanced security of our system. Although being useful as it provides a larger connection pool, the RDS proxy is too pricey for our deployment, thus we've resigned from it.
Authentication	Password	The most basic option.
Deletion protection	Yes	Minimizing the risk of unwanted data loss.
Retention period	7 days	A basic retention period is enough for our applications.

Table 3 Amazon RDS Configuration describes the most important parameters and options.

Security

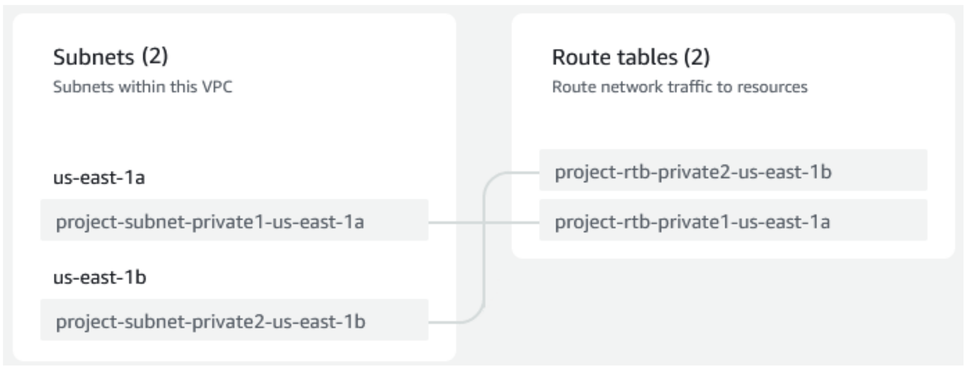
The security of our application mostly depends on the security of the database, thus we will describe it in this subsection. Firstly, as mentioned in the previous subsection, the database is available only from our AWS VPC, with no public access, which already makes it harder to commit a serious data breach. Additionally, the instanced are placed inside of the private subnets without the ability to connect to the outside world. To protect it from unauthorized access, we've also ensured the SQL injection protection within the application code, and the passwords placed in the storage are hashed using the bcrypt algorithm. Each argument to the query has to be a single valid sql value, checked before sending the query to the database. Thus we prevent any unplanned queries from being performed on our data. Another layer of protection is limiting the access to the data only for people that are supposed to have it, thus we've secured a master account with a password and created a security group for managing the databases. Finally, we've turned on the deletion protection to prevent accidental data loss.

Backup and Disaster Recovery

Despite securing the data, we also have to provide high availability of our service, thus we had to ensure it with a backup system and disaster recovery strategies. The major pillar of these actions depends on the Multi-AZ deployment of both the databases and AWS Lambda instances. With this solution, we can ensure that the users won't lose their data, as the standby database will be always available in a separate availability zone. Thanks to the Multi-AZ technology, when the failover happens the application will automatically change to the standby instance with the current data. Moreover, according to the AWS documentation, the mean time between switching to the standby instance takes up to 1 minute. We could improve all these pillars by adding a third AZ, which would improve the availability, and even reduce the RTO to around 40 seconds. Another option is deploying the application in another region, doubling the costs. In case of bigger budgets, we would probably decide to combine both strategies.

VPC configuration

Project-vpc is the singular vpc in our project with API Gateway being the only service outside of it. It covers the security of our app and database in two availability zones. During development there were additional services such as Internet gateway and public subnet, deleted after the app has been finished.


Subnets (2)
Subnets within this VPC
us-east-1a
project-subnet-private1-us-east-1a
us-east-1b
project-subnet-private2-us-east-1b
Route tables (2)
Route network traffic to resources
project-rtb-private2-us-east-1b
project-rtb-private1-us-east-1a

User manual

A not logged in user on our website has two options, namely login and registering. A new user has to use the register fields, inputting a username and password which are over 5 characters in length. After that login is possible. Stored passwords are hashed, so administrators of the database can't access users passwords.

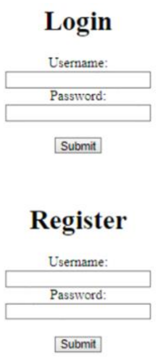
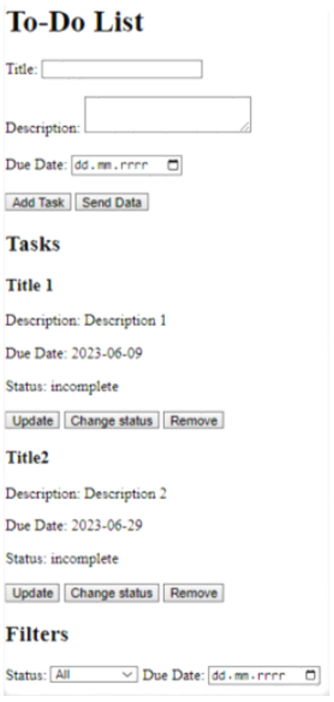


Figure 4 Login/register screen.

After logging in a new view appears. It consists of a field for inputting new to-do tasks and a list of our current tasks. To input a new task we fill the title, due date and the description of it in corresponding fields. After adding it (Add Task), the data shows up in our list of tasks. There we can update the status or delete the task. Clicking update opens up a pop-up that allows us to change all the fields in a task. Clicking "Change status" switches between the 3 possible statuses. If we want to select some of the tasks, we can filter them by completion status and/or the due date. When all edits have been made, we can finalize the updates by clicking "Send Data" which uploads the data to the database.


To-Do List

Title:

Description:

Due Date: dd.mm.rrrr

Tasks

Title 1

Description: Description 1

Due Date: 2023-06-09

Status: incomplete

Title2

Description: Description 2

Due Date: 2023-06-29

Status: incomplete

Filters

Status: All Due Date: dd.mm.rrrr

Figure 55 Tasks screen after successful login.

Unit Tests

The tests were performed using the unittest library from Python. We have created two classes *TestDatabase* responsible for database related tests and *TestEndpoints* for endpoint tests.
In the *TestDatabase* class, the following tests are performed:

1. *test_create_connection*: This test ensures that a connection to the PostgreSQL database can be successfully established by trying to connect using the connection string. If the connection is successful, the test passes.
2. *test_user*: This test focuses on the user-related operations in the database. It first tries to insert a user with a test username and password. If the insertion is successful, it retrieves the user from the database and verifies that the result contains exactly inserted user. Then, it attempts to delete the user from the database.
3. *test_tasks*: This test verifies the functionality related to tasks in the database. It starts by inserting a user with a test username and password, and obtaining a token from cookie. Then, it deletes any existing tasks associated with the token. After that, it inserts a test task into the database using the token. The test then retrieves the tasks associated with the token and checks if the result contains exactly one task. Finally, it deletes the tasks and the user from the database.

In the *TestEndpoints* class, the following tests are performed:

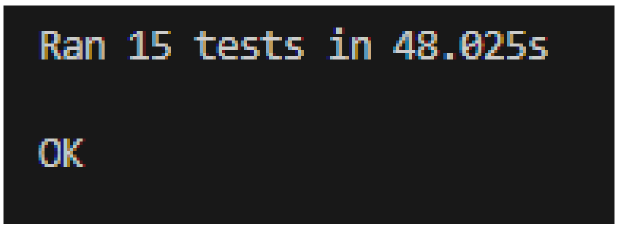
1. *test_get*: This test sends a GET request to the root endpoint ("/") of the application and checks if the response status code is 200, indicating a successful request.
2. *test_get_login*: This test sends a GET request to the "/login" endpoint and checks if the response status code is 200.
3. *test_get_register*: This test sends a GET request to the "/register" endpoint and checks if the response status code is 200.
4. *test_get_list*: This test performs a series of operations: it first registers a user with a test username and password,

then logs in using the same credentials, and finally sends a GET request to the "/list" endpoint. The test verifies that the response status code is 200. After the test, it deletes the user from the database.

5. *test_post_register_invalid*: This test sends a POST request to the "/register" endpoint with an invalid username and password and checks if the response status code is 401, indicating unauthorized access.
6. *test_post_register_existing*: This test sends a POST request to the "/register" endpoint with an existing username and password and checks if the response status code is 401.
7. *test_post_register:* This test sends a POST request to the "/register" endpoint with a correct test username and password and checks if the response status code is 200. After the test, it deletes the user from the database.
8. *test_post_login_invalid*: This test first registers a user with a test username and password, then sends a POST request to the "/login" endpoint with the correct username but an incorrect password. It checks if the response status code is 401. After the test, it deletes the user from the database.
9. *test_post_login_nonexisting*: This test first registers a user with a test username and password, then sends a POST request to the "/login" endpoint with a random username and password. It checks if the response status code is 401. After the test, it deletes the user from the database.
10. *test_post_login_token*: This test first registers a user with a test username and password, then sends a POST request to the "/login" endpoint with the correct username and password. It checks if the response status code is 302, indicating a redirect, and if a token is present in the response cookies. After the test, it deletes the user from the database.
11. *test_post_login*: This test first registers a user with a test username and password, then sends a POST request to the "/login" endpoint with the correct username and password. It checks if the response status code is 200. After the test, it deletes the user from the database.
12. *test_post_list*: This test performs a series of operations: it first registers a user with a test username and password, then logs in using the same credentials, and finally sends a POST request to the "/list" endpoint with a JSON payload containing a test task. The test verifies that the response status code is 200. After the test, it deletes the user from the database.

Application passed all above tests.

.


```
Ran 15 tests in 48.025s

OK
```

Tips and Tricks

In this section, we want to briefly summarize what have we learned during the implementation of this project, outside of the main scope of the course.

1. **Database Creation:**
    a. If you are using RDS solutions, during the early development select one AZ, and no backup to limit the costs, as Multi-AZ is easily applicable later on and RDS solutions generate big constant costs,
    b. For the sake of cost optimization, prefer to use the no-sql databases, as they are billed mostly for access

to the data, and don't have any constant costs,

   c. During the early development make the database public, so you can easily connect from pgAdmin. As your database is empty at this stage, nothing bad can happen, although remember to change it as soon as any data arrives,
   d. Don't use the RDS proxy, it enhances the costs greatly for every instance, including another region and standby copies.

2. **Lambda Application:**
   a. Develop the app using VPCs Endpoint Gateway, so the AWS services are easily accessible,
   b. Firstly, develop the app, and later manage the cloud, as you can easily witness if you break something,
   c. Create one Lambda with multiple endpoints for small applications, it's less work while adding new features.
3. **Cloudcraft Diagrams:**
   a. It's a useful tool to use during the planning phase,
   b. Greatly helps with the cost estimation,
   c. Reminds about placing the services inside VPC, and subnets, everything is visible from a single dashboard.

Table of tables

Table of Figures