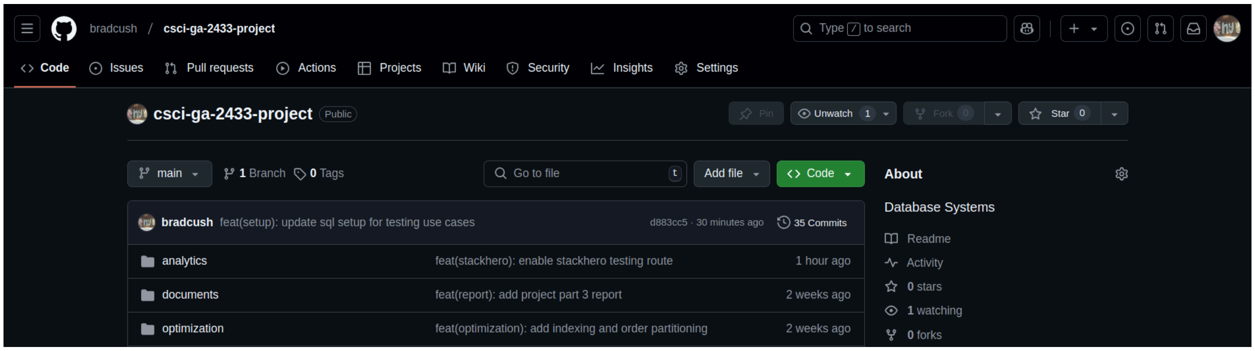
Name: Cushing, Bradley
Date: 12/17/24
NYU ID: N10695516
Course Section Number: CSCI-GA-2433-001
Project Part #4 Report

End-to-End Solution Design and Implementation

Public GitHub link: https://github.com/bradcush/csci-ga-2433-project



bradcush feat(setup): update sql setup for testing use cases d883cc5 · 30 minutes ago 35 Commits

```
analytics      feat(stackhero): enable stackhero testing route          1 hour ago
documents      feat(report): add project part 3 report                  2 weeks ago
optimization   feat(optimization): add indexing and order partitioning  2 weeks ago
```

Note that everything that was created to produce parts 1-4 of the project are contained in the above public GitHub repository. This includes but is not limited to images, screenshots, previous reports, source code, DDL physical/logical schemas, conceptual schemas, etc. All files referenced in this report document are located in previously submitted reports or in the GitHub repository linked above which I briefly give an overview of below.

Repository files

There are a lot of files and folders in the repository so here's a high level description of all of them to help you better navigate through the code base if you would like to. All relevant files and folders to the work in this Project Part 4 are referenced in their respective sections as well.

- /analytics (scripts using S3 automating uploading data and training models)
- /documents (final documents submitted for each part of the project along the way)
- /optimization (separate SQL files containing physical schema optimizations)
- /prediction (early scripts used to train/predict logistic regression model)
- /queries (some SQL files to help during development and testing)
- /routes (code for routes implementing all of the use cases)
- /schema (DDL files and conceptual schemas at each iteration)
- /screenshots (all screenshots used across all project reports)
- app.py (entry point for web application importing routes)
- README.md (development setup documentation)

1. Select, design, and implement business use cases

Live web application: https://glacial-ocean-75196-9463e01a29ef.herokuapp.com/

The above link is to the final implementation for all use cases that were defined in Project Part 2 and Project Part 3 and slightly modified in this Project Part 4 for the fictitious Circuit Blocks, Inc business I created. I'll briefly review these use cases in section 2 of this report as they relate to what's been implemented. Please refer to section 3 and section 4 of this report for a deeper dive on the exact implementation details.

2. Document your business use cases

As mentioned above, see Project Part 3 for documentation around use cases. Project Part 3 also contains the diagram outlining the design for my data-driven workflow-based application but I've included it here and briefly mentioned how it relates to what's implemented.

Use case documentation

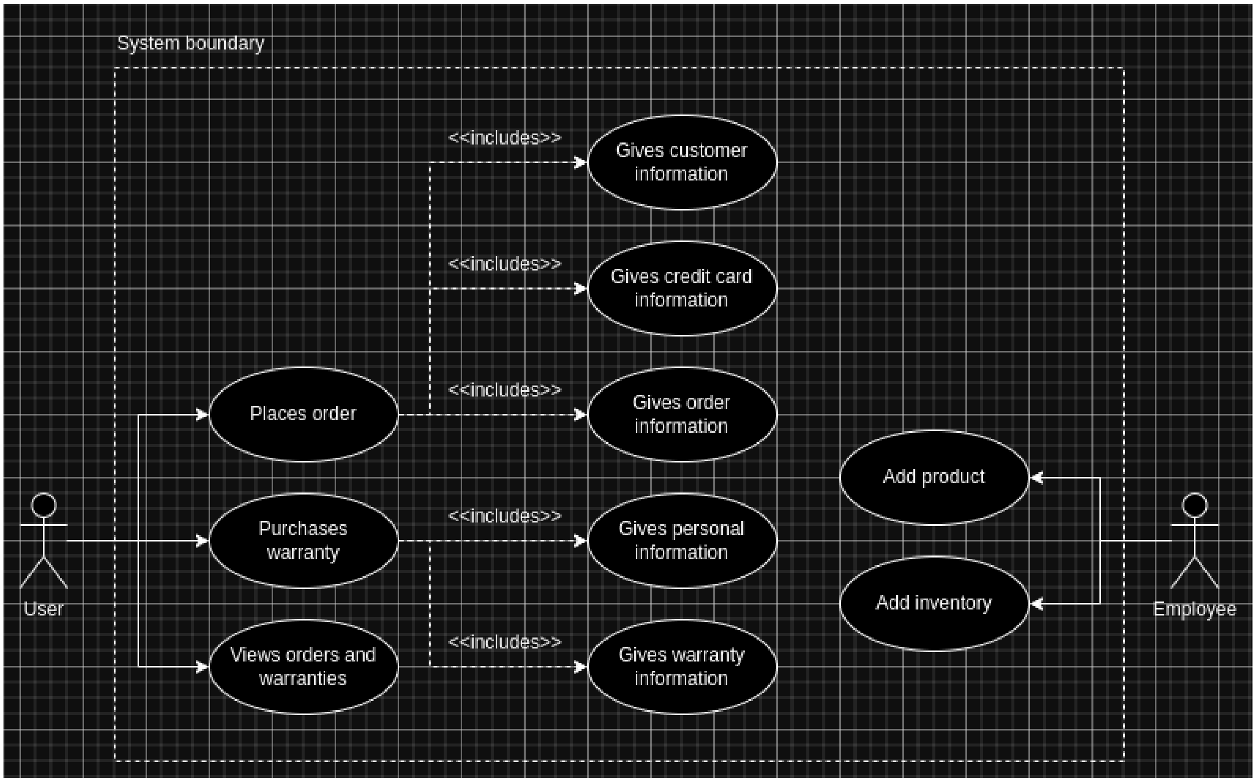
The above link to the production end-to-end application lists 8 use cases that are part of a single end-to-end workflow, some steps to be executed by employees and others by customers. First, an employee can enter a product into the database and subsequently, its associated inventory. Once the above steps have been completed, a user can add themselves by signing up as a customer, add credit card information, place an order, enter their personal information, and purchase a warranty.

Note that the purchasing of the warranty is where we use our prediction model which was previously trained to come up with a price and percentage based on the predicted risk of the customer. Lastly, customers can view warranties, which shows some relevant information from each table for a user and their related warranties if any exist.

- Employee enters product
- Employee updates inventory
- User enters customer details
- User provides credit card information
- User provides order information
- User submits personal information
- User provides warranty information
- User views warranty information

Use case diagram

The below use case diagram was copied from the previous Project Part 3. It shows all use cases mentioned above using UML notation to more clearly illustrate things.



Data-driven workflow-based application

The below diagram and design have evolved starting from Project Part 1 and through to Project Part 3. Project Part 3 should be referenced for the most up-to-date design but I'll give a high-level recap here to make things clear. We've separated the overall application into three sections: being in business, staying in business, and external.

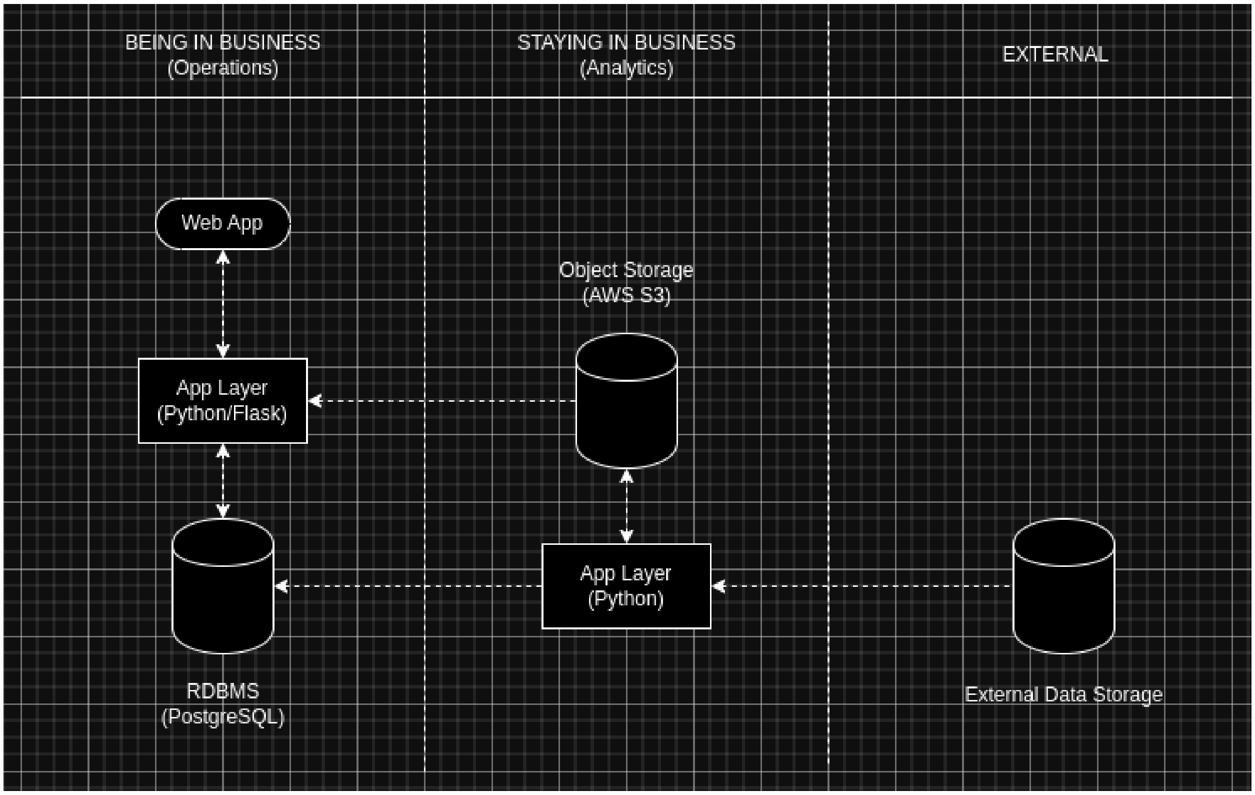
Being in business

The part powers the operational side of the business. When an employee or user wants to execute a use case, they navigate to the Web App, fill out a form with some data, and submit the form. This Web App is a Python/Flask application served from the App Layer. When a form is submitted, some checks are made on the data and what exists in the database already. If everything is successful, the App Layer adds this data to the RDBMS database layer.

When personal information is added, we make use of the connection between the Operations side and the Analytics side. We fetch the pre-trained model from the storage on the Analytics side and calculate their risk associate for that personal information, entering it in the RDMS on the Operations side. This allows us to easily calculate the percentage and price for the warranty if a warranty is purchased at a later time.

Staying In Business

Our design includes support for re-training the model when external data has been updated. We have storage for the external data used to train a model in an S3 bucket. This data is updated periodically and in an ideal world the App Layer has a job that runs to check if there are any new data sets that we should use to train. If we find one, we retrain the model on that data set and save the model in the Object Storage which is another S3 bucket. We can easily make sure we retrieve the latest model from the Operations side each time we make a prediction.



I've implemented several things for automatically dealing with new external data sets and retraining the model to be used for risk assessment when new external data exists. Remember that previously we decided to use Amazon S3 for storing both external data and trained models. There are two scripts listed below which for now are meant to be run locally to help automate this process. The general ideal is that we upload new external data to S3 when it becomes available. We later use this data which is on S3 to retrain a model. After that we can simply download the latest model every time we need to make a prediction. This approach also makes sure to archive old external data and models which were used previously.

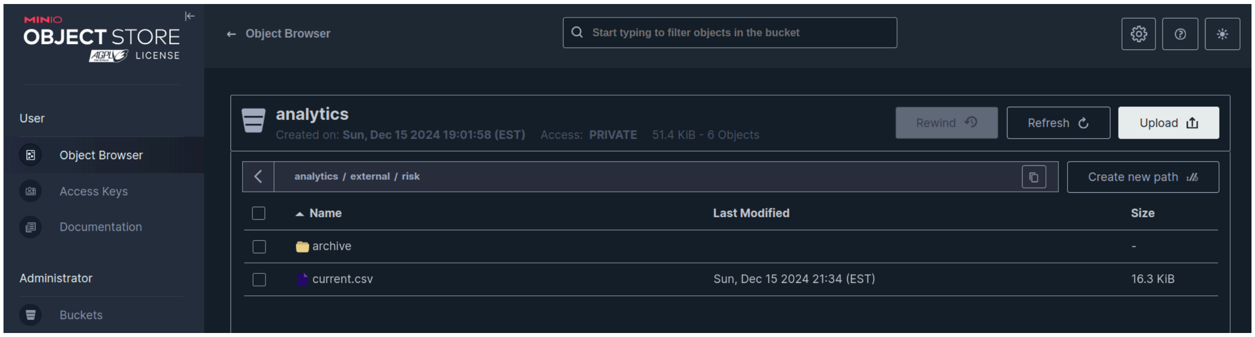
- analytics/upload_risk.py
- analytics/train_latest.py

Uploading external data

For now uploading new external data can be done using the script “analytics/upload_risk.py”. We expect the current CSV data to be in the same folder and labeled “current.csv”. When the script is run, we first move the old “current.csv” file on S3 to the archive folder in the S3 bucket “analytics/external/risk” with a new name which is the hashed time at the time it was archived.

We also move the file locally into an archive folder with the same hash that was used for remote. This will allow us to retrieve the most current external data from S3 in a predictable location when we need to train a new model which I explain in the next section.

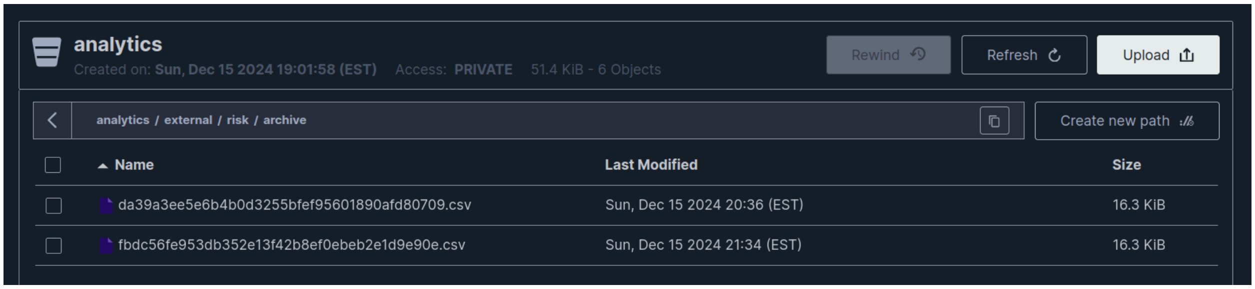
Since this data should change infrequently, this approach should be good for our needs. In the future if we wanted to do this automatically this script could be run on a remote server like proposed in our final design above and combined with a job that fetches the latest external data from the original source it comes from. I've included two screenshots below to show these files which have been uploaded and versioned in our S3 storage bucket.



analytics
Created on: Sun, Dec 15 2024 19:01:58 (EST) Access: PRIVATE 51.4 KiB - 6 Objects

analytics / external / risk

Name	Last Modified	Size
archive		-
current.csv	Sun, Dec 15 2024 21:34 (EST)	16.3 KiB



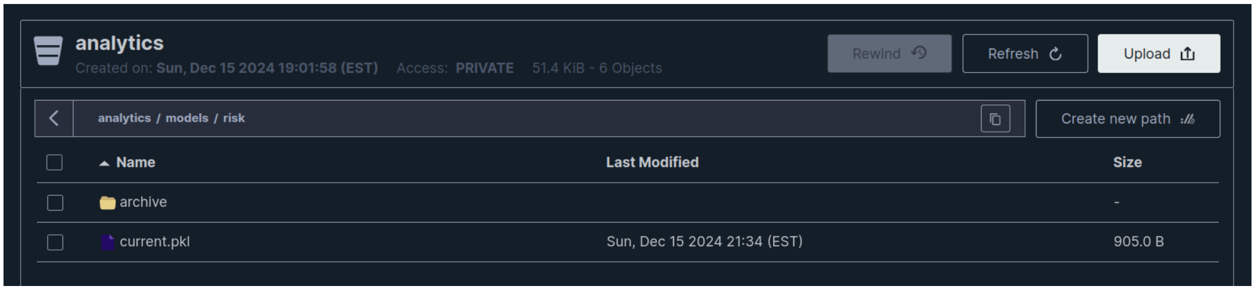
analytics
Created on: Sun, Dec 15 2024 19:01:58 (EST) Access: PRIVATE 51.4 KiB - 6 Objects

analytics / external / risk / archive

Name	Last Modified	Size
da39a3ee5e6b4b0d3255bfef95601890afd80709.csv	Sun, Dec 15 2024 20:36 (EST)	16.3 KiB
fbdc56fe953db352e13f42b8ef0ebeb2e1d9e90e.csv	Sun, Dec 15 2024 21:34 (EST)	16.3 KiB

Training the model

Similar to the process above, we have a script that can be run locally for the time being to train a model when new data is uploaded to our remote S3 bucket. If there is a new "current.csv" file on S3, the script "analytics/train_latest.py" fetches this file, trains the model locally using this data and subsequently saves the trained model, a "pkl" file, in another S3 bucket "analytics/models/risk". We save this model as "current.pkl" similar to the procedure in the previous section and archive the model we're replacing by hashing the current time like above and moving it into an archive folder. I've attached screenshots below showing the admin panel view of the S3 bucket after running this script below.

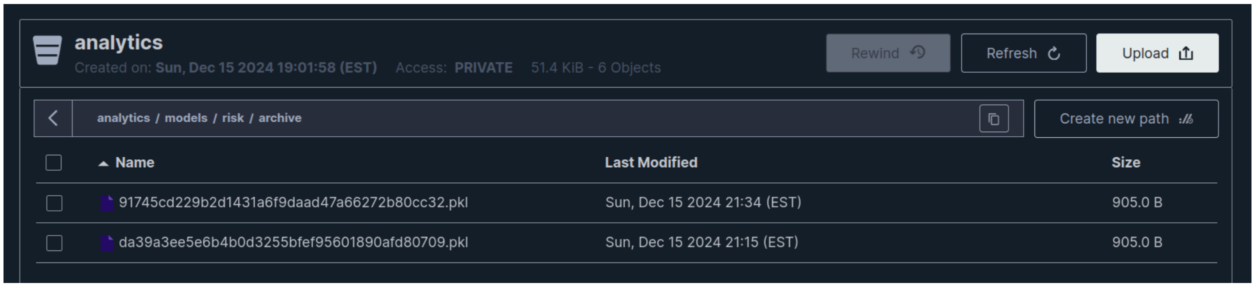


analytics

Created on: Sun, Dec 15 2024 19:01:58 (EST)   Access: PRIVATE   51.4 KiB - 6 Objects

analytics / models / risk

Name	Last Modified	Size
archive		-
current.pkl	Sun, Dec 15 2024 21:34 (EST)	905.0 B



analytics

Created on: Sun, Dec 15 2024 19:01:58 (EST)   Access: PRIVATE   51.4 KiB - 6 Objects

analytics / models / risk / archive

Name	Last Modified	Size
91745cd229b2d1431a6f9daad47a66272b80cc32.pkl	Sun, Dec 15 2024 21:34 (EST)	905.0 B
da39a3ee5e6b4b0d3255bfef95601890afd80709.pkl	Sun, Dec 15 2024 21:15 (EST)	905.0 B

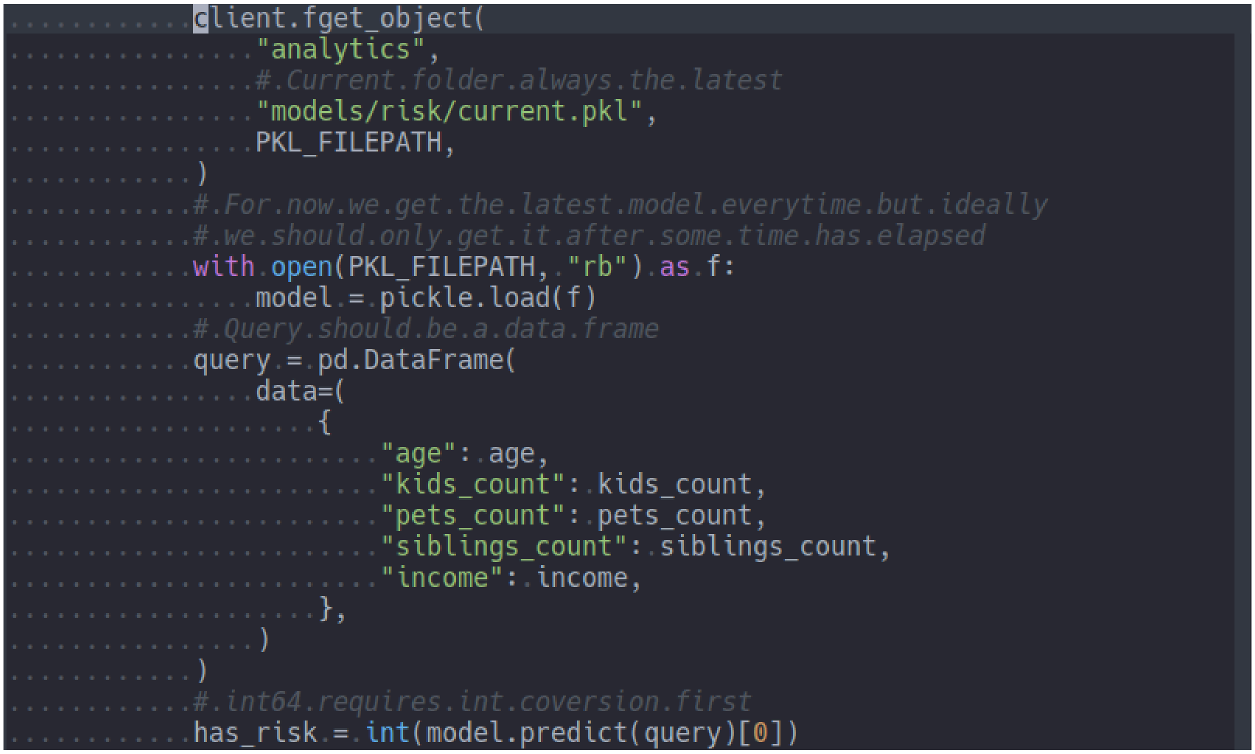
Fetching the model

Remember that when the user enters personal information, we need to assess the risk to the business for a warranty based on it. In our current setup, we simply download the latest model from S3 in the location we expect, “analytics/model/risk/current.pkl” and run inference with the downloaded model created using the steps in the previous section. Currently we download the model each time we need to run inference so we always have an updated model. This is a bit expensive and in the future we could do this at some fixed interval.

3. Create a data-driven program module

This part of the application was built using Python/Flask, psycopg2, sklearn, and pickle. Python/Flask is the general programming language and framework for serving a web application. psycopg2 is a PostgreSQL database adapter for Python that was used to connect to and interact with the RDBMS which we chose to be PostgreSQL. sklearn is an open-source library for predictive analysis we used to train and run the mode. pickle is a library that allows us to save the trained model in a “pki” format and later load the model to run it. Looking more closely at the use case “User submits personal information”, this is where we retrieve and run the machine learning model that has been trained against data that is entered from the user. Specifically in the file “routes/add_personal.py” file does this.

For this route in our application, we can navigate to “Enter personal information”, submit personal information in the form, and see the resulting personal_information table in the RDBMS if the entry is successfully added. I’ll briefly state here what happens when information is submitted in the form and “routes/add_personal.py” is executed. I’ve also included a partial screenshot below including some of the code for this endpoint.

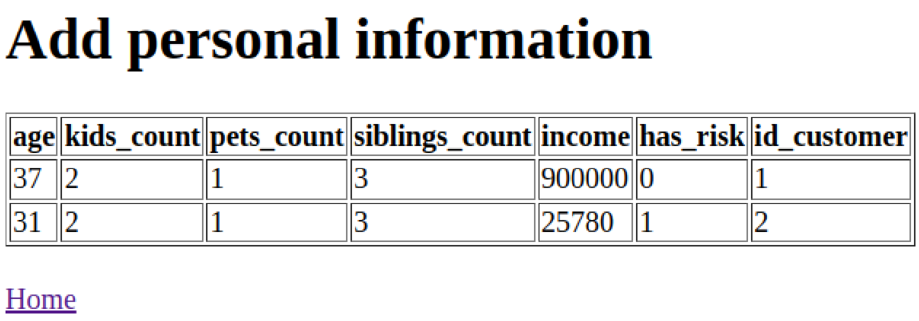


```
            client.fget_object(
                "analytics",
                # Current folder always the latest
                "models/risk/current.pkl",
                PKL_FILEPATH,
            )
            # For now we get the latest model everytime but ideally
            # we should only get it after some time has elapsed
            with open(PKL_FILEPATH, "rb") as f:
                model = pickle.load(f)
            # Query should be a data frame
            query = pd.DataFrame(
                data=(
                    {
                        "age": age,
                        "kids_count": kids_count,
                        "pets_count": pets_count,
                        "siblings_count": siblings_count,
                        "income": income,
                    },
                )
            )
            # int64 requires int coversion first
            has_risk = int(model.predict(query)[0])
```

This route reads query parameters submitted from the form containing the personal information of the user. We then fetch the latest model from the local file system, where in our ideal architecture above would be stored in an S3 bucket. We run the prediction using the model and the data from the query parameters to get a has_risk prediction for the user. Then we finally insert this information into the database, overwriting any previous entry that may have existed for this user already, so only the latest information is valid. Making information overwritable made it easier to test this functionality and update predictions. If the personal_information table is updated successfully we show the entries from this table to the user. I've included a screenshot below of the resulting table when testing.

Links to enter data into and view the personal_information table:

- https://glacial-ocean-75196-9463e01a29ef.herokuapp.com/enter-personal
- https://glacial-ocean-75196-9463e01a29ef.herokuapp.com/add-personal



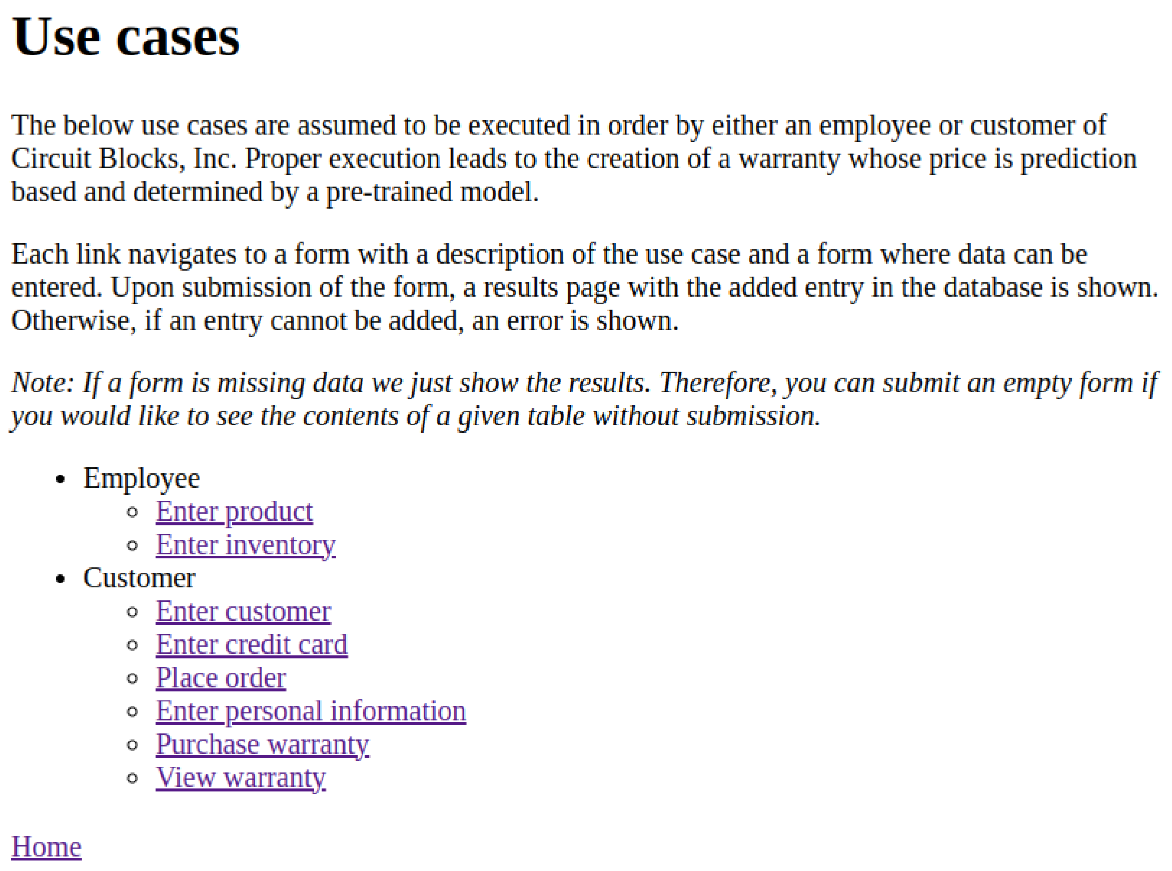
Add personal information

age	kids_count	pets_count	siblings_count	income	has_risk	id_customer
37	2	1	3	900000	0	1
31	2	1	3	25780	1	2

Home

4. Database connectivity framework

I already covered how we integrate with the prediction model in section 3, here I'll discuss how the rest of what was implemented. Like the above, this part of the application was also built using Python/Flask and psycopg2. I've included a link below to the homepage which has links to all use cases. Each link goes to a form where information can be entered into the respective tables according to the case cases specified in the previous sections. The last link is just for viewing information related to warranties if all information has been entered using the above links. I've also attached a screenshot of the homepage for convenience.



Use cases

The below use cases are assumed to be executed in order by either an employee or customer of Circuit Blocks, Inc. Proper execution leads to the creation of a warranty whose price is prediction based and determined by a pre-trained model.

Each link navigates to a form with a description of the use case and a form where data can be entered. Upon submission of the form, a results page with the added entry in the database is shown. Otherwise, if an entry cannot be added, an error is shown.

Note: If a form is missing data we just show the results. Therefore, you can submit an empty form if you would like to see the contents of a given table without submission.

- Employee
  - Enter product
  - Enter inventory
- Customer
  - Enter customer
  - Enter credit card
  - Place order
  - Enter personal information
  - Purchase warranty
  - View warranty

Home

Homepage link with links to all use cases:
https://glacial-ocean-75196-9463e01a29ef.herokuapp.com/

Please use the link above to view the use cases online. Note that if you would like to view any data that is present in the tables, data that has previously been added, you can submit an empty form which will bring you to a result page showing all table entries. I've already populated the database with some information, going through the above use cases with several fictitious users manually to ensure everything is implemented correctly.

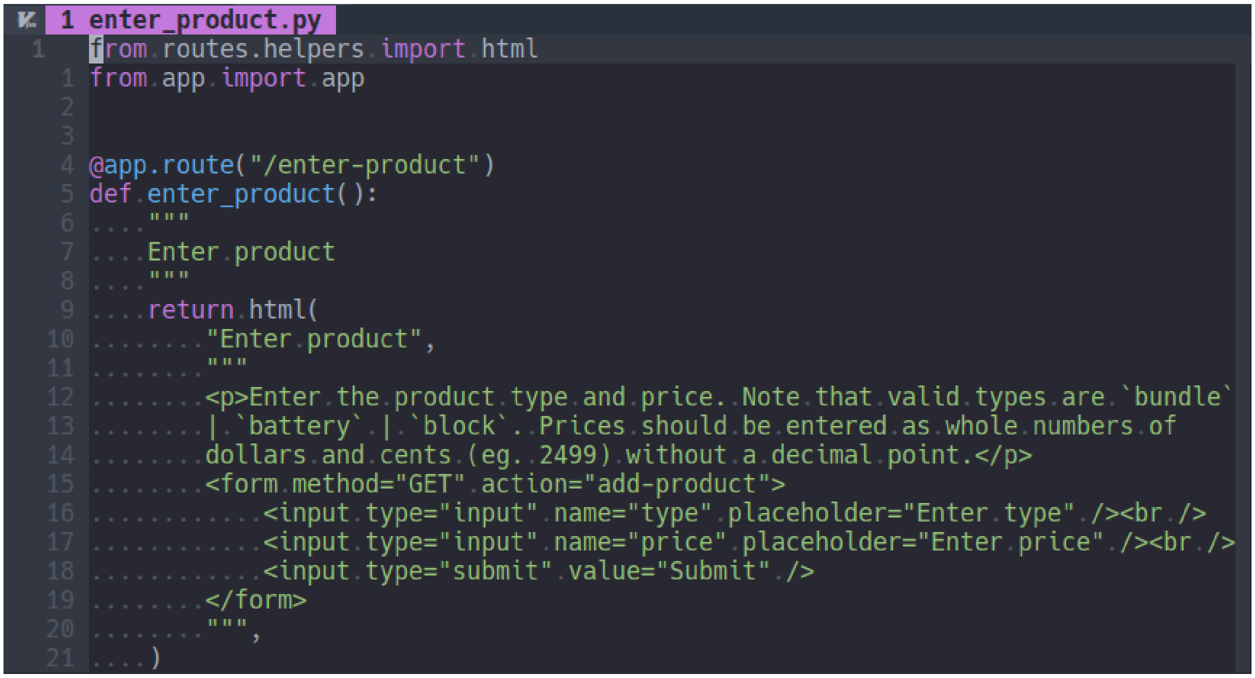
Note: It may take a couple seconds to load on a first visit. I'm using a Heroku dev environment which requires time to start up on a first visit if the service has been idle for some time.

Relevant source code

Source code is organized by route, where most routes are prefixed with either “enter” or “add”, which are for entering data into forms or adding data based on the submitted forms respectively. There are only 3 routes that don’t fit this pattern, specifically “place_order.py”, “purchase-warranty.py”, and “view-warranty.py”.

- routes/add_credit_card.py
- routes/add_customer.py
- routes/add_inventory.py
- routes/add_order.py
- routes/add_personal.py
- routes/add_product.py
- routes/add_warranty.py
- routes/enter_credit_card.py
- routes/enter_customer.py
- routes/enter_inventory.py
- routes/enter_personal.py
- routes/enter_product.py
- routes/home.py
- routes/place_order.py
- routes/purchase_warranty.py
- routes/view_warranty.py

Below is a screenshot of what the code looks like for one of the forms, specifically this is the form for entering a product into the database use case. For code related to other endpoints and processing of data listed above, please view the source code in the routes folder in the GitHub repository referenced at the top of this report.



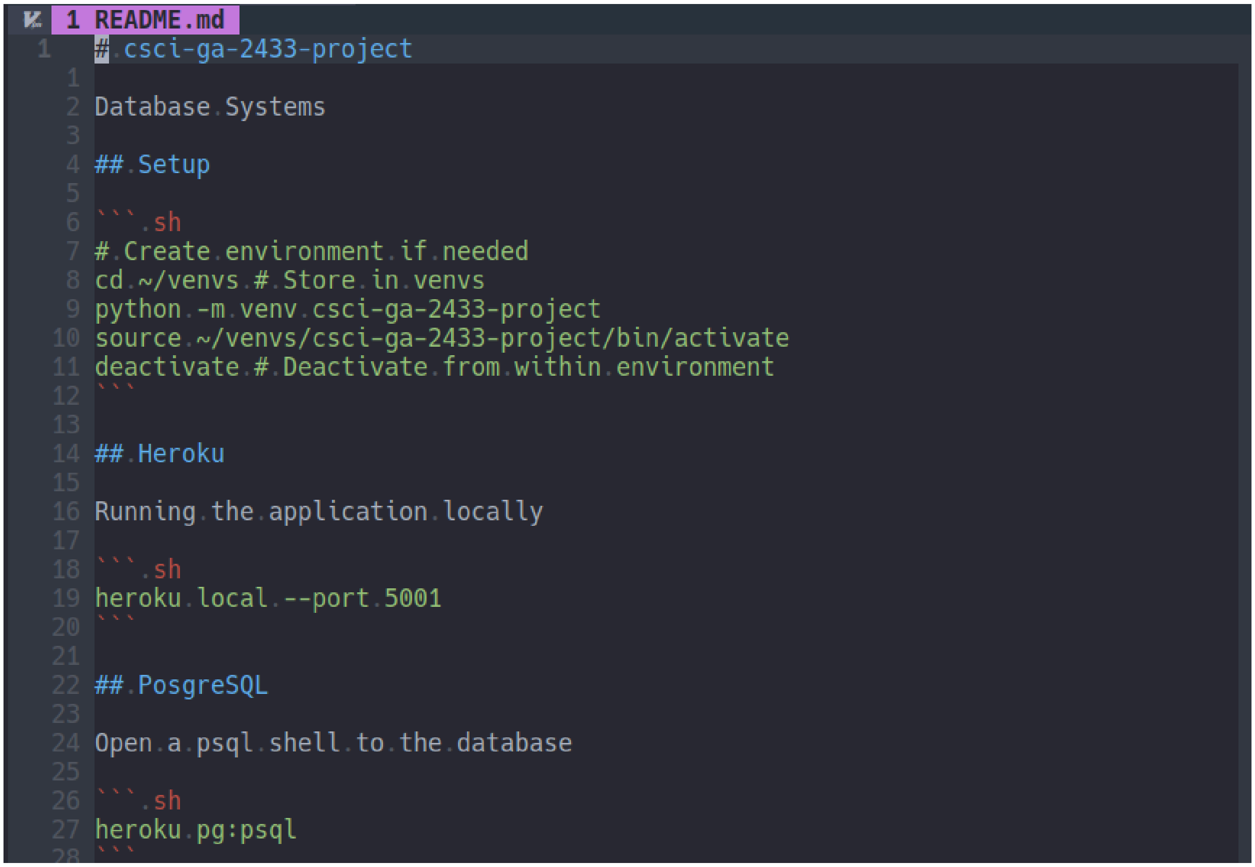
```python
from routes.helpers import html
from app import app


@app.route("/enter-product")
def enter_product():
    """
    Enter product
    """
    return html(
        "Enter product",
        """
        <p>Enter the product type and price.  Note that valid types are `bundle`
        | `battery` | `block`. Prices should be entered as whole numbers of
        dollars and cents (eg. 2499) without a decimal point.</p>
        <form method="GET" action="add-product">
            <input type="input" name="type" placeholder="Enter type" /><br />
            <input type="input" name="price" placeholder="Enter price" /><br />
            <input type="submit" value="Submit" />
        </form>
        """,
    )
```

Use case screenshots

I encourage you to visit the link I mentioned previously to see all use cases but I've included screenshots in the next section to highlight a few of them and show that everything is working as intended. Note that there are 8 use cases I've implemented and 7 of them have two views (enter, add) so this list of screenshots is quite exhaustive.

5. Document your application

First I've included a README.md file which explains how to set up and run the project from a developer perspective. Local testing is difficult if you don't have a PostgreSQL database running locally but the instructions are there nonetheless. Below is a screenshot that shows the beginning of this README.md file. Please reference the original for more information.



~~~
# csci-ga-2433-project

Database Systems

## Setup

```sh
# Create environment if needed
cd ~/venvs # Store in venvs
python -m venv csci-ga-2433-project
source ~/venvs/csci-ga-2433-project/bin/activate
deactivate # Deactivate from within environment
~~~

Heroku

Running the application locally

```sh
heroku local --port 5001
```

PosgreSQL

Open a psql shell to the database

```sh
heroku pg:psql
```
```

Application screenshots

There are also many screenshots in Project Part 3 that show the application running both locally on my machine and remotely. I’ve included all screenshots below for all 8 use cases that were implemented, showing them in the order they should be executed. All screenshots below are of the fully functioning webapp and were taken after running through each use case with specific data that made sense for that use case.
```

# Use cases

The below use cases are assumed to be executed in order by either an employee or customer of Circuit Blocks, Inc. Proper execution leads to the creation of a warranty whose price is prediction based and determined by a pre-trained model.

Each link navigates to a form with a description of the use case and a form where data can be entered. Upon submission of the form, a results page with the added entry in the database is shown. Otherwise, if an entry cannot be added, an error is shown.

*Note: If a form is missing data we just show the results. Therefore, you can submit an empty form if you would like to see the contents of a given table without submission.*

- Employee
  - Enter product
  - Enter inventory
- Customer
  - Enter customer
  - Enter credit card
  - Place order
  - Enter personal information
  - Purchase warranty
  - View warranty

Home

# Enter product

Enter the product type and price. Note that valid types are \`bundle\` | \`battery\` | \`block\`. Prices should be entered as whole numbers of dollars and cents (eg. 2499) without a decimal point.

Enter type

Enter price

Submit

Home

# Add product

| id | type | price |
|---|---|---|
| 1 | bundle | 2499 |
| 2 | battery | 499 |
| 3 | block | 199 |

Home

# Enter inventory

Inventory includes a location, quantity, and valid product id. Any location and quantity is allowed but the `id_product` value must be the valid id of an existing product in the database.

Enter location

Enter quantity

Enter id_product

Submit

Home

# Add inventory

| location | quantity | id_product |
|---|---|---|
| poughkeepsie | 500 | 1 |
| poughkeepsie | 250 | 2 |
| poughkeepsie | 1000 | 3 |
| berkeley | 200 | 1 |
| berkeley | 100 | 2 |
| berkeley | 500 | 3 |

Home

# Enter customer

Enter the name, email, and phone to create a new customer. Each customer is automatically assigned a distinct identifier by the system. The `id` will be used to identify a specific user when entering data into other forms related to other use cases.

Enter name

Enter email

Enter phone

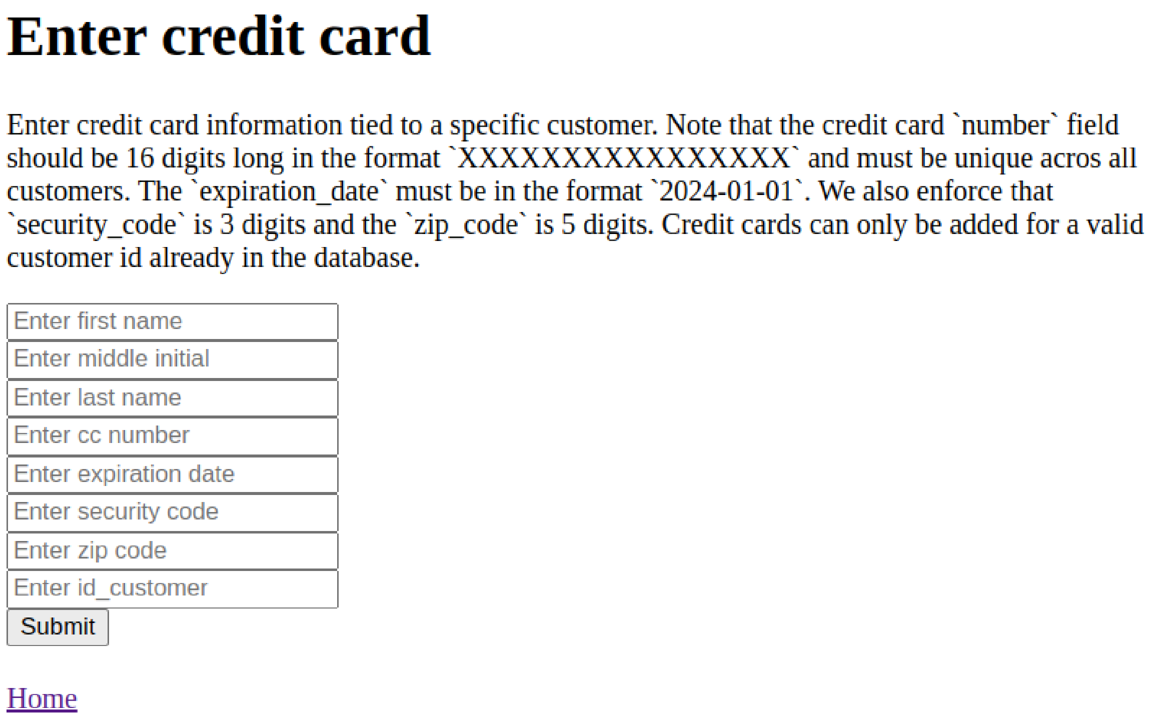
Submit

Home

# Add customer

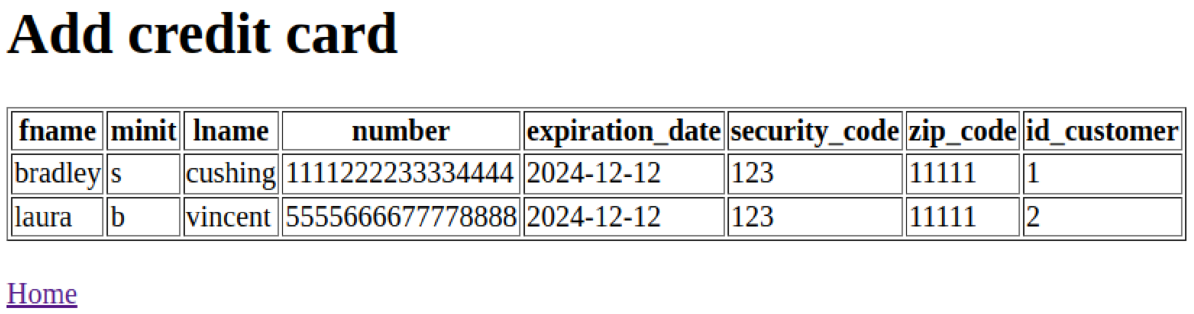
| id | name | email | phone |
|---|---|---|---|
| 1 | bradley | bradley@whatit.be | 1234567890 |
| 2 | laura | laura@whatit.be | 1234567890 |
| 3 | oodie | oodie@whatit.be | 1234567890 |

Home



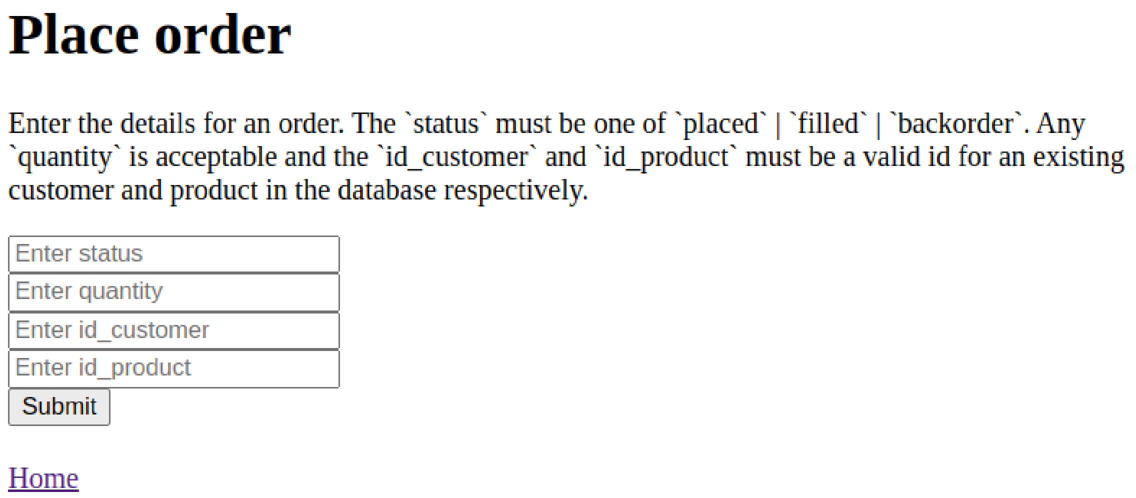
# Enter credit card

Enter credit card information tied to a specific customer. Note that the credit card `number` field should be 16 digits long in the format `XXXXXXXXXXXXXXXX` and must be unique acros all customers. The `expiration_date` must be in the format `2024-01-01`. We also enforce that `security_code` is 3 digits and the `zip_code` is 5 digits. Credit cards can only be added for a valid customer id already in the database.



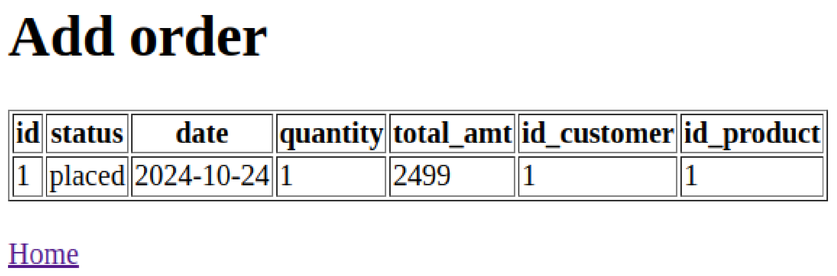
# Add credit card

| fname | minit | lname | number | expiration_date | security_code | zip_code | id_customer |
|---|---|---|---|---|---|---|---|
| bradley | s | cushing | 1111222233334444 | 2024-12-12 | 123 | 11111 | 1 |
| laura | b | vincent | 5555666677778888 | 2024-12-12 | 123 | 11111 | 2 |



# Place order

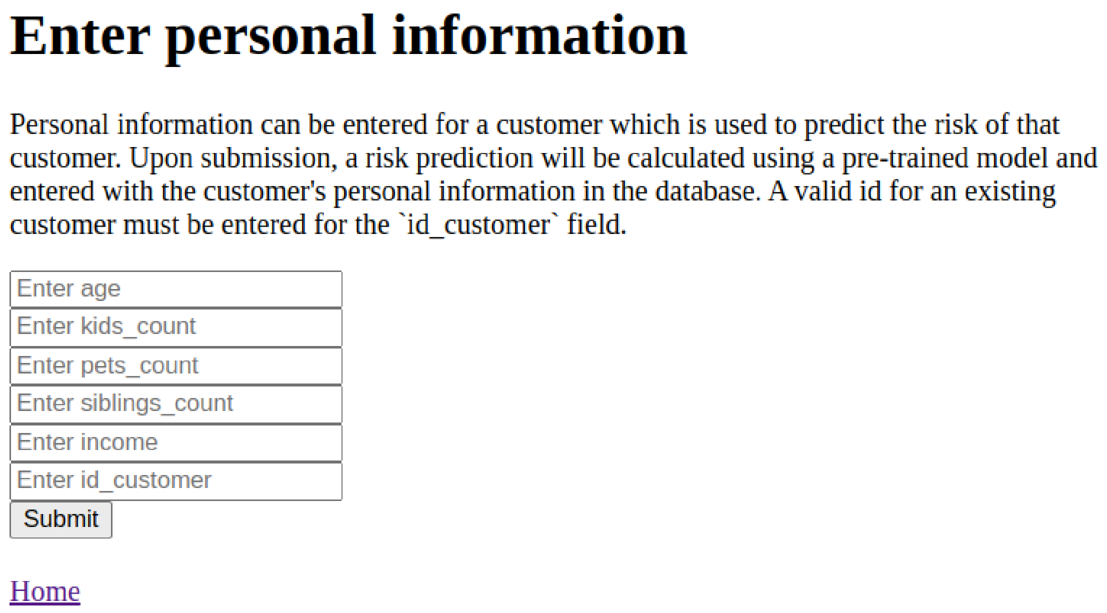
Enter the details for an order. The `status` must be one of `placed` | `filled` | `backorder`. Any `quantity` is acceptable and the `id_customer` and `id_product` must be a valid id for an existing customer and product in the database respectively.



# Add order

| id | status | date | quantity | total_amt | id_customer | id_product |
|---|---|---|---|---|---|---|
| 1 | placed | 2024-10-24 | 1 | 2499 | 1 | 1 |

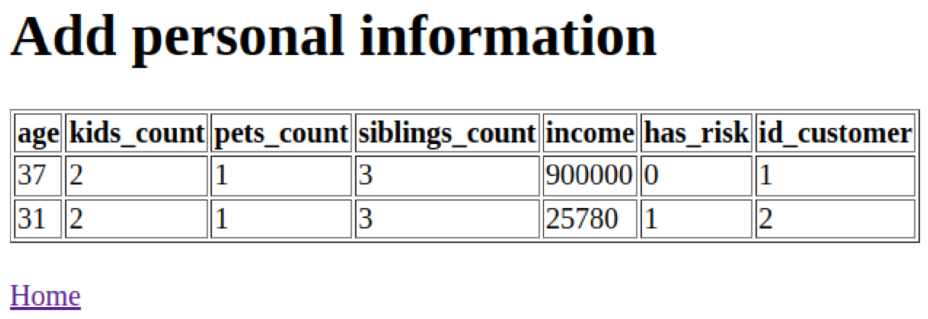
Home



# Enter personal information

Personal information can be entered for a customer which is used to predict the risk of that customer. Upon submission, a risk prediction will be calculated using a pre-trained model and entered with the customer's personal information in the database. A valid id for an existing customer must be entered for the `id_customer` field.

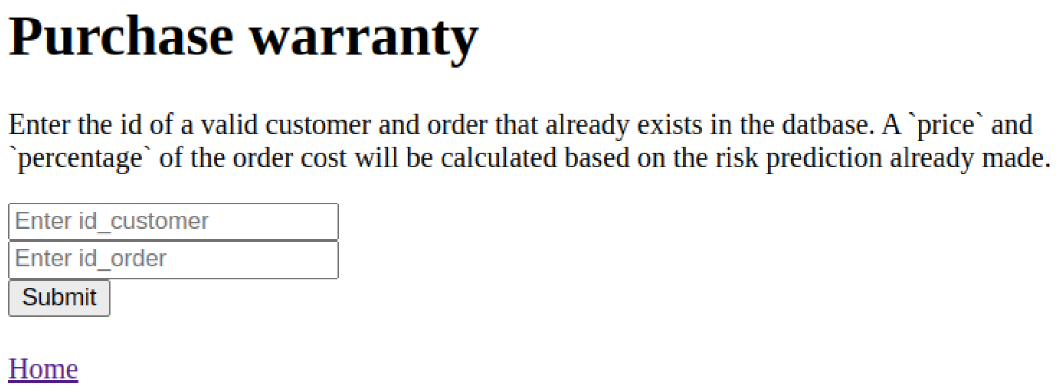
Home



# Add personal information

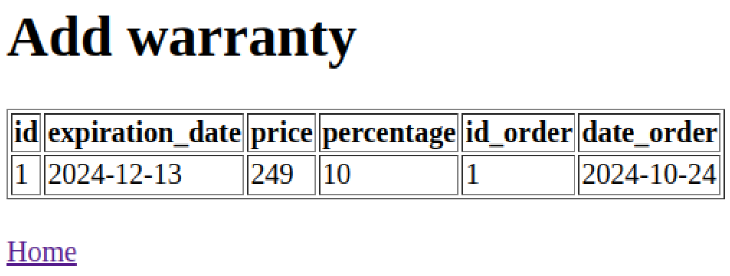
| age | kids_count | pets_count | siblings_count | income | has_risk | id_customer |
|---|---|---|---|---|---|---|
| 37 | 2 | 1 | 3 | 900000 | 0 | 1 |
| 31 | 2 | 1 | 3 | 25780 | 1 | 2 |

Home



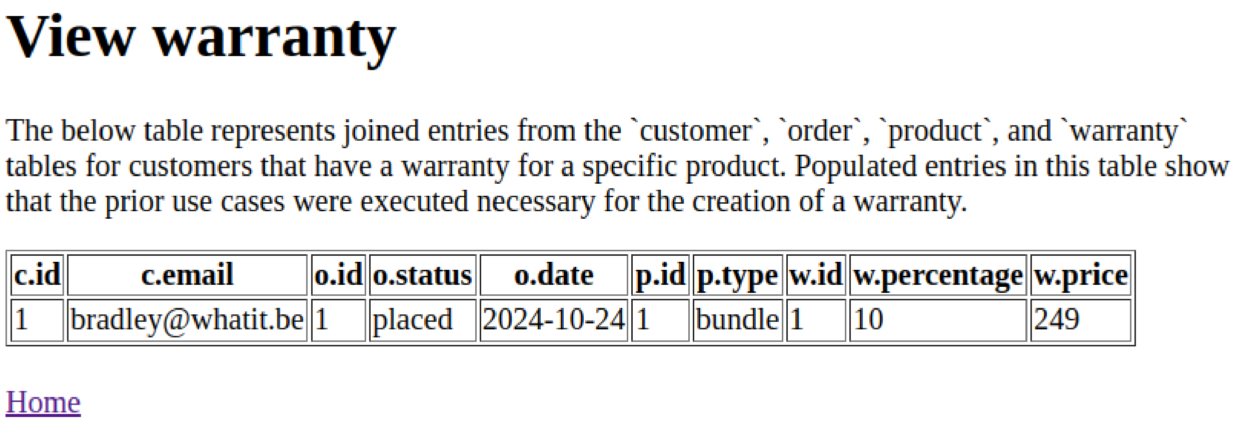
# Purchase warranty

Enter the id of a valid customer and order that already exists in the datbase. A `price` and `percentage` of the order cost will be calculated based on the risk prediction already made.



# Add warranty

| id | expiration_date | price | percentage | id_order | date_order |
|---|---|---|---|---|---|
| 1 | 2024-12-13 | 249 | 10 | 1 | 2024-10-24 |



# View warranty

The below table represents joined entries from the `customer`, `order`, `product`, and `warranty` tables for customers that have a warranty for a specific product. Populated entries in this table show that the prior use cases were executed necessary for the creation of a warranty.

| c.id | c.email | o.id | o.status | o.date | p.id | p.type | w.id | w.percentage | w.price |
|---|---|---|---|---|---|---|---|---|---|
| 1 | bradley@whatit.be | 1 | placed | 2024-10-24 | 1 | bundle | 1 | 10 | 249 |

### Meeting the requirements

This application meets the requirements specified for the project by implementing 8 different use cases that can all be combined from beginning to end to allow a customer to order a product and purchase a warranty for that order. It provides a fully functional web interface that is interactive and can be used to execute every part of the workflow needed.

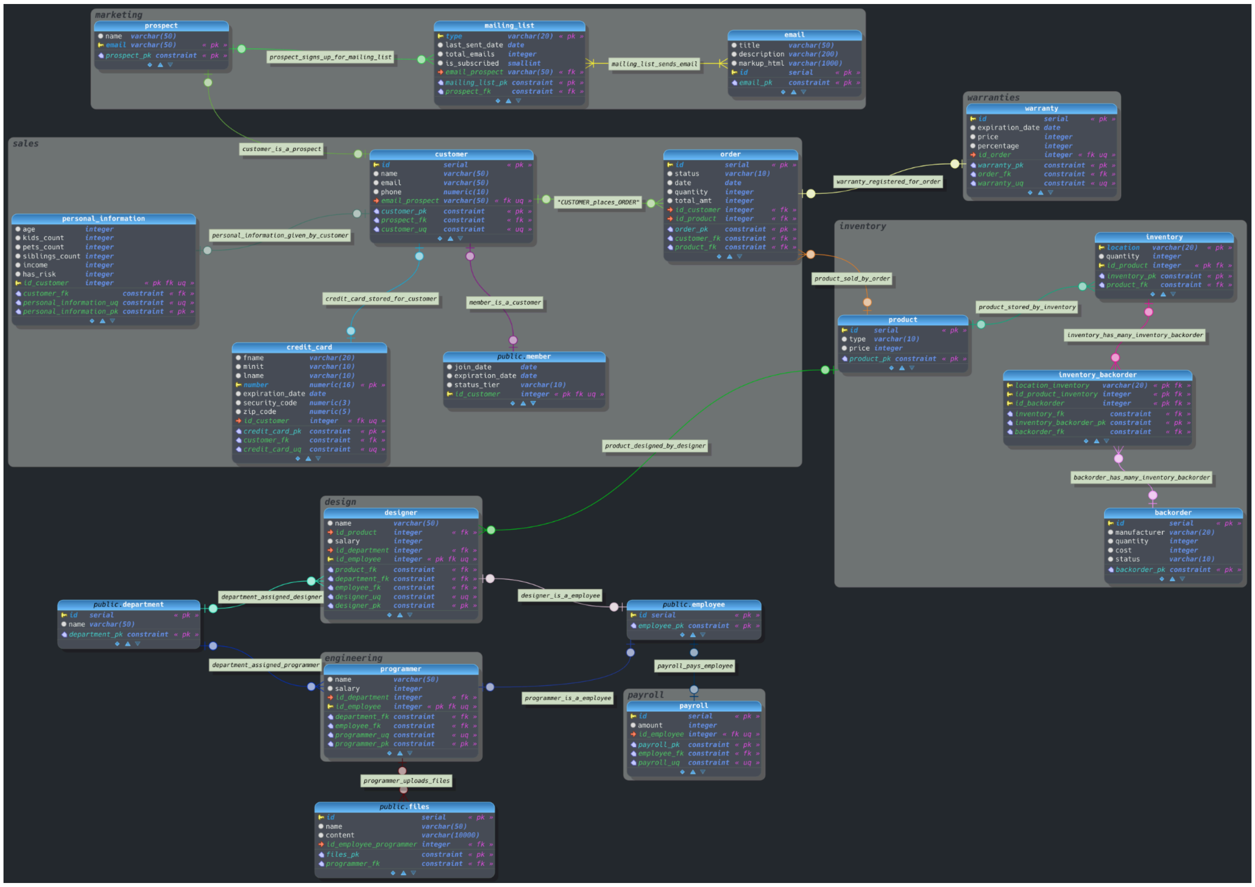
Furthermore, this project trains a machine learning model (logistic regression) to make useful predictions about customer risk pertaining to warranties in real time based on external data. All of the examples provided are real examples generated from using this application and model and can be verified by visiting the GitHub link and public link to the application.

## Optimization techniques

All optimization techniques used for the physical model of the database like indexing and partitioning have also been leveraged. Please see Project Part 3 for all details related to optimization including why specific choices were made and their benefits.

In regard to optimization of the logical model, complete normalization was done in Project Part 2 which I reference here. All normalization techniques are described thoroughly in that section. All DDL SQL commands for generating the schema and optimizations are also included as part of Project Part 2 and refined further in Project Part 3 after indexing and partitioning were added.

I've also included a screenshot below of the latest conceptual model which has been updated to reflect categorization across departments of the business. It's a little small as it includes 19 different entities but you can reference the original file in the code repository labelled "schema/circuit-blocks-4.png". The more general schema without this categorization update is referenced in Project Part 2. Note that all updates are after normalization was applied which can also be referenced in Project Part 2.


marketing
prospect
name varchar(50)
email varchar(50) « pk »
prospect_pk constraint « pk »
prospect_signs_up_for_mailing_list
mailing_list
type varchar(20) « pk »
last_sent_date date
total_emails integer
is_subscribed smallint
email_prospect varchar(50) « fk »
mailing_list_pk constraint « pk »
prospect_fk constraint « fk »
mailing_list_sends_email
email
title varchar(50)
description varchar(200)
markup_html varchar(1000)
id serial « pk »
email_pk constraint « pk »
warranties
warranty
id serial « pk »
expiration_date date
price integer
percentage integer
id_order integer « fk uq »
warranty_pk constraint « pk »
order_fk constraint « fk »
warranty_uq constraint « uq »
sales
customer_is_a_prospect
customer
id serial « pk »
name varchar(50)
email varchar(50)
phone numeric(10)
email_prospect varchar(50) « fk uq »
customer_pk constraint « pk »
prospect_fk constraint « fk »
customer_uq constraint « uq »
"CUSTOMER_places_ORDER"
order
id serial « pk »
status varchar(10)
date date
quantity integer
total_amt integer
id_customer integer « fk »
id_product integer « fk »
order_pk constraint « pk »
customer_fk constraint « fk »
product_fk constraint « fk »
warranty_registered_for_order
personal_information
age integer
kids_count integer
pets_count integer
siblings_count integer
income integer
has_risk integer
id_customer integer « pk fk uq »
customer_fk constraint « fk »
personal_information_uq constraint « uq »
personal_information_pk constraint « pk »
personal_information_given_by_customer
credit_card_stored_for_customer
member_is_a_customer
credit_card
fname varchar(20)
minit varchar(10)
lname varchar(10)
number numeric(16) « pk »
expiration_date date
security_code numeric(3)
zip_code numeric(5)
id_customer integer « fk uq »
credit_card_pk constraint « pk »
customer_fk constraint « fk »
credit_card_uq constraint « uq »
public.member
join_date date
expiration_date date
status_tier varchar(10)
id_customer integer « pk fk uq »
product_designed_by_designer
inventory
product_sold_by_order
product_stored_by_inventory
inventory
location varchar(20) « pk »
quantity integer
id_product integer « pk fk »
inventory_pk constraint « pk »
product_fk constraint « fk »
inventory_has_many_inventory_backorder
product
id serial « pk »
type varchar(10)
price integer
product_pk constraint « pk »
inventory_backorder
location_inventory varchar(20) « pk fk »
id_product_inventory integer « pk fk »
id_backorder integer « pk fk »
inventory_fk constraint « fk »
inventory_backorder_pk constraint « pk »
backorder_fk constraint « fk »
backorder_has_many_inventory_backorder
backorder
id serial « pk »
manufacturer varchar(20)
quantity integer
cost integer
status varchar(10)
backorder_pk constraint « pk »
design
designer
name varchar(50)
id_product integer « fk »
salary integer
id_department integer « fk »
id_employee integer « pk fk uq »
product_fk constraint « fk »
department_fk constraint « fk »
employee_fk constraint « fk »
designer_uq constraint « uq »
designer_pk constraint « pk »
department_assigned_designer
designer_is_a_employee
public.department
id serial « pk »
name varchar(50)
department_pk constraint « pk »
public.employee
id serial « pk »
employee_pk constraint « pk »
department_assigned_programmer
payroll_pays_employee
engineering
programmer
name varchar(50)
salary integer
id_department integer « fk »
id_employee integer « pk fk uq »
department_fk constraint « fk »
employee_fk constraint « fk »
programmer_uq constraint « uq »
programmer_pk constraint « pk »
programmer_is_a_employee
payroll
payroll
id serial « pk »
amount integer
id_employee integer « fk uq »
payroll_pk constraint « pk »
employee_fk constraint « fk »
payroll_uq constraint « uq »
programmer_uploads_files
public.files
id serial « pk »
name varchar(50)
content varchar(10000)
id_employee_programmer integer « fk »
files_pk constraint « pk »
programmer_fk constraint « fk »

## 6. Finalize end-to-end reference architecture

Here I restate the reference architecture used for this project. I'll first restate the vision and core principles for this project. Then I'll give insight into the information architecture and how the pieces we have developed fit into the broader picture of what's been defined for this type of business. The Reference Architecture slides were used as a model when creating this architecture so we have taken some of our inspiration from there.

### Vision

The vision for the Circuit Blocks Inc. reference architecture to enable the efficient building and scaling of information and technology that power the Circuit Blocks, Inc. business and allow the company to serve customers in the best way possible.

### Principles

- All technology choices should be chosen based on the business strategy
- The reference architecture should guide all decisions making when possible
- Architecture decisions should be categorized into two buckets: those decisions which are easy to revert to their original state, those that are hard to revert to their original state
- All decisions that are hard to revert must be approved by the architecture committee
- Regardless of the effort to revert we require all high cost project get approval
- Those decisions that are easy to revert can be taken within a department
- When choosing a technology we should consider the business as a whole
- Teams should be able to work independently of one another as much as possible
- If an existing solution solves a problem well, consider it before inventing a new one
- Every project or piece of technology needs to have a single owner in the business
- If a system becomes so large that one team can't own it, it should be split so that there is an ownership boundary and multiple independent owners for those parts.
- Team and individual autonomy is encouraged in decision making
- We promote taking action by doing and asking for forgiveness later
- The reference architecture should constantly evolve with the business

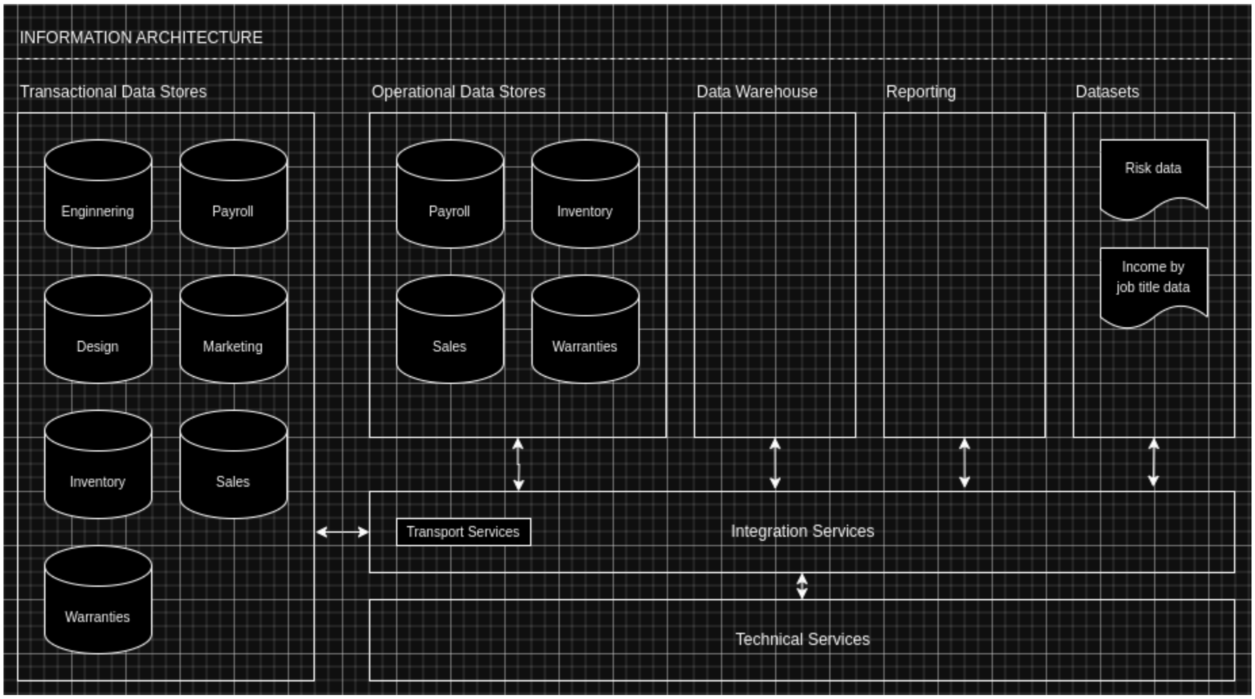
### Information architecture

We're considering a few additional departments compared to what was specified in Project Part 1. All departments accounted for in our conceptual diagram based on the entities we have.

#### Departments

Engineering
Design
Marketing
Inventory
Warranties
Payroll
Sales

The following information architecture diagram explains the information architecture visually for the Circuit Blocks, Inc. business. We reference everything that we currently support regarding what was built or envisioned from the beginning. We also leave room for areas we hope to support in the future but currently aren't part of our business. For this reason we have included the Data Warehouse, Reporting, and Technical Services sections even though we don't have those functions at the moment.



### Transactional Data Stores

This is our PostgreSQL relational database which holds all of the data for our business we require to do business and serve our customers. Each of the different departments is represented in this section, both customer data as well as internal data.

### Transport Service

In this diagram we make a distinction between Transactional Data Stores and Operational Data Stores. In our current case we only have one relational database which is used for both but we model what it would look like if we were to copy data we wanted to query over to the operational side. This requires a Transport Service which is why we've included it here.

### Datasets

We have two external Datasets mentioned here, the Risk data which we are currently using in our system to make predictions and the Income by job title data which we proposed at the beginning and could potentially use in the future. This data is processed as is by the model but we could imagine cleaning and sanitizing it in the Technical Services layer first.

### Policies and Guidelines

- This information model should be update whenever changes are made
- Information should not be duplicated by different departments
- Transactional Data Stores should never be queried for reporting
- All reporting should be made available across all of the departments
- Integration Services and Technical Services should be shared cross functions
- Physical schemas should be optimized based on queries made by departments
- Physical schemas should be reviewed and updated once a quarter
- Logical schemas should be reviewed with plans for updating yearly
- Data constraints on entered and stored data should be as strict as possible
- The architecture committee must sign off on all proposed logical schema updates
- All external datasets should be updated at regular, specified intervals
- Python is the primary and preferred language for information tasks. If another language is required it should be checked with the architecture committee before using it.
- PostgreSQL should be used as the only RDBMS solution
- Amazon S3 should be used for all  BLOB storage

## Deliverables

- Public GitHub link: https://github.com/bradcush/csci-ga-2433-project
- Live web application: https://glacial-ocean-75196-9463e01a29ef.herokuapp.com/