

DROP TABLE Software_Engineering ATC Project Documentation

Phil Dysert
Carlos Santana
Joanna Al Madanat Jaar
Hudson Gieck
Heather Bowes

Introduction

The project for this course was to create a system from start to finish following the software development life using the tools we learned about in class and drawing upon what each student has learned in each class before this one. The purpose of this project was twofold both to learn the software development life cycle and to learn how to work on a software engineering team. The deliverables were means to this end.

- Accomplishments by Team Member
 - List significant project components authored/completed
 - List total hours worked
 - List documentation component by author
- Heather
 - Significant project components completed: Infrastructure
 - VPC
 - ECR
 - EKS
 - Docker
 - CircleCI
 - Kubernetes
 - Docker
 - Ansible
 - Hours - 83.13
 - Documentation sections
 - Exempt from documentation due to her significant contributions to the project.
- Jo
 - Significant project components completed: Code
 - Unit tests
 - CRUD
 - Airline collision detection
 - Authentication
 - Duplicate Gate/Runway
 - Kafka (& undoing Kafka because she got it working)
 - Hours - 126.11
 - Documentation sections
 - Wrote a lot of the Code Architecture section
 - **Exempt from (more) documentation due to the ridiculous # of hours spent coding.**
- Hudson
 - Significant project components completed:
 - Burndown charts
 - FMEA analysis
 - Hours - 63.05

- Documentation sections
 - Project Design
- Phil
 - Significant project components completed:
 - SCRUM master (sprints 2-4)
 - Static Assets transferred to S3 Bucket
 - Route 53 Domain configured
 - FTA analysis
 - Hours - 62.36
 - Documentation sections:
 - Introduction
 - Project Requirements
- Carlos
 - Significant project components completed:
 - Data loading
 - Prometheus
 - Hours - 32.47
 - Documentation sections
 - Evently
 - App Interaction

Initial Project Requirements

Build a system capable of monitoring air traffic between multiple airports that can create, read, update, and delete:

- Users (gate agents and air traffic controllers)
- Airplanes
- Airports
- Airlines
- Runways
- Gates

Updated Project Requirements After Analysis

The system shall:

- Compare the gate size to the plane size when assigned and reject the assignment if the gate size is smaller than the plane size.
- Compare the runway size to the plane size when assigned and reject the assignment if the runway size is smaller than the plane size.
- Check if a gate is already assigned to a plane and reject further assignments while the gate is assigned.
- Check if a runway is already assigned to a plane and reject further assignments while the runway is assigned.
- Compare the airport schedule to the plane's schedule and reject the landing assignment if the airplane is not scheduled at that airport.
- Compare the airplane size to the number of passengers and reject passenger assignment if airplane size is smaller.

Project Design (Hudson)

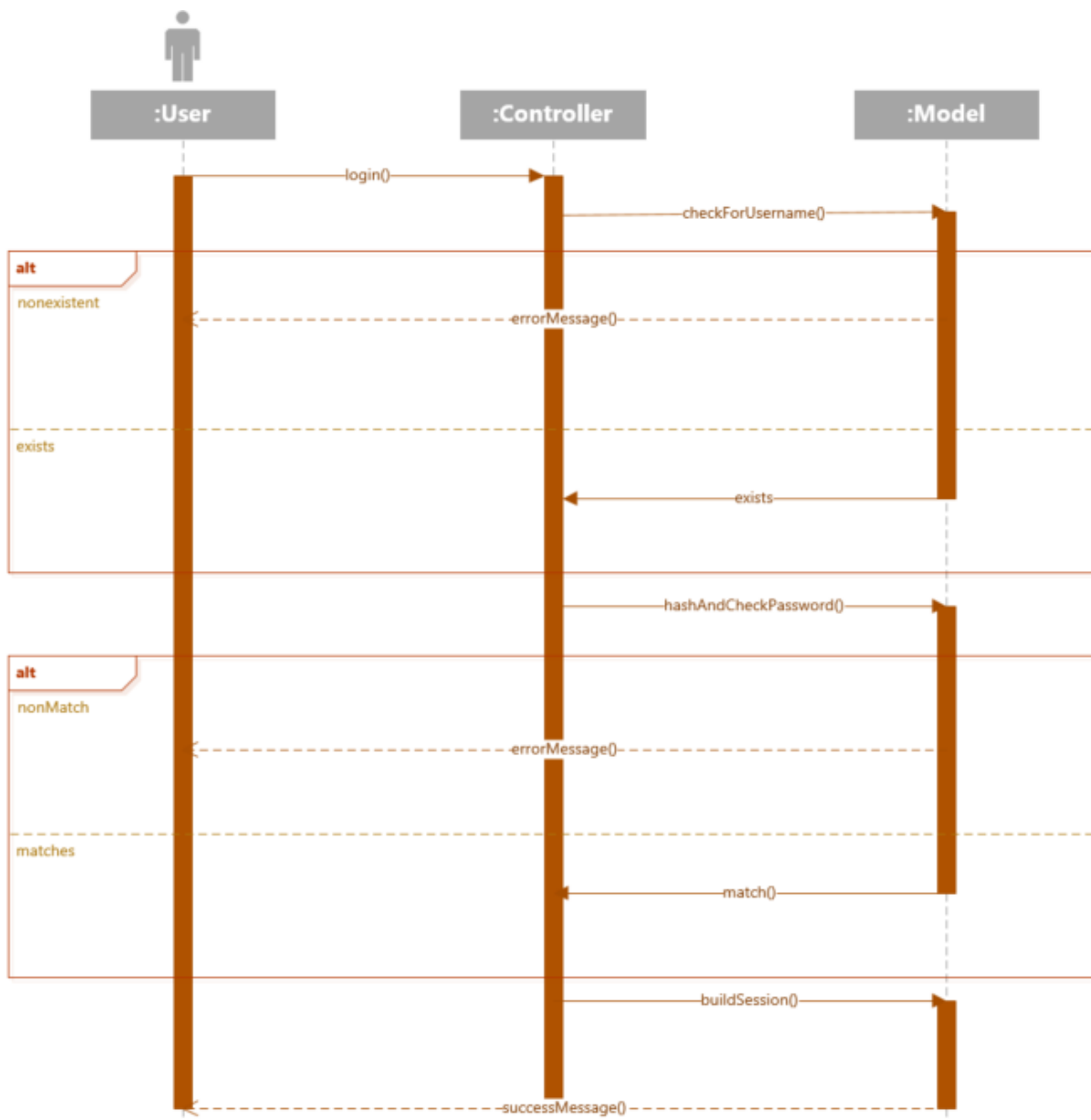
Design Overview

- Login checks username, password, and hash before accepting or denying connection
- Logout checks if session exists or not before determining to end the connection
- Create checks for unique data before validating and then persisting the data in the database
- Update checks for existing data before validating and then persisting the updated data in the database
- Delete checks for existing data before deleting it from the database
- Risk assessment shows catastrophic failures across the board, but the design of the application makes the failures unlikely

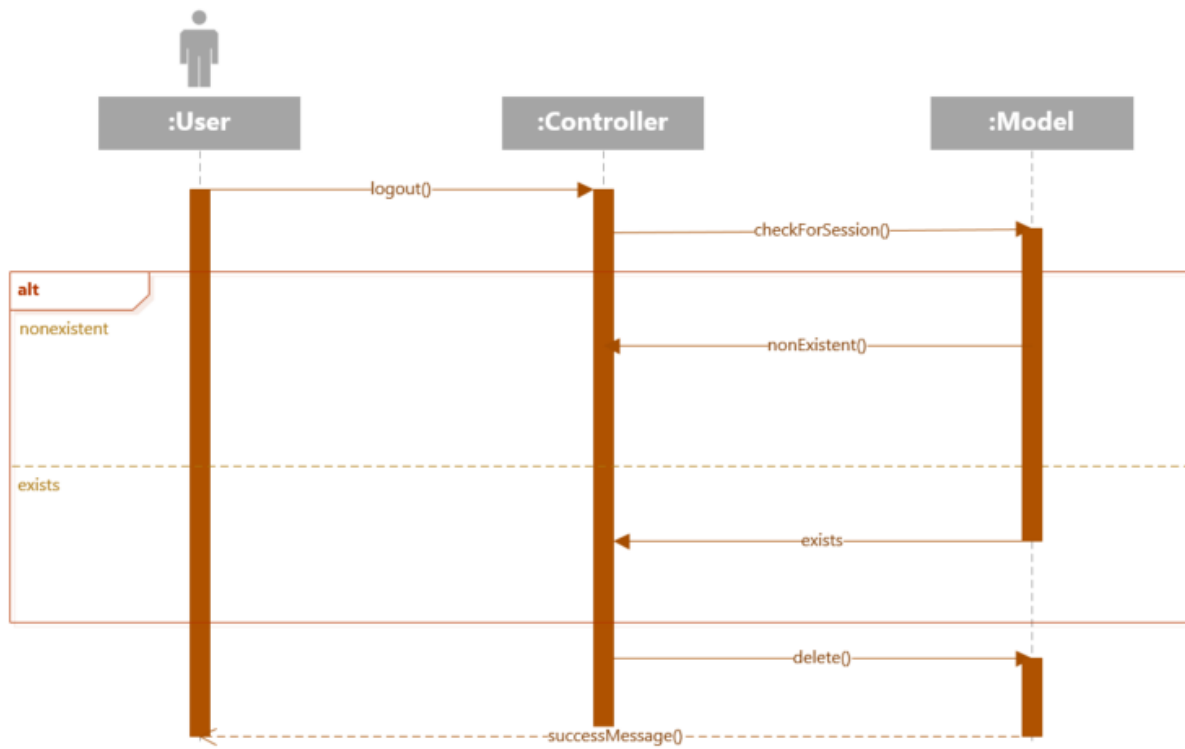
Updated Figures

Sequence Diagrams

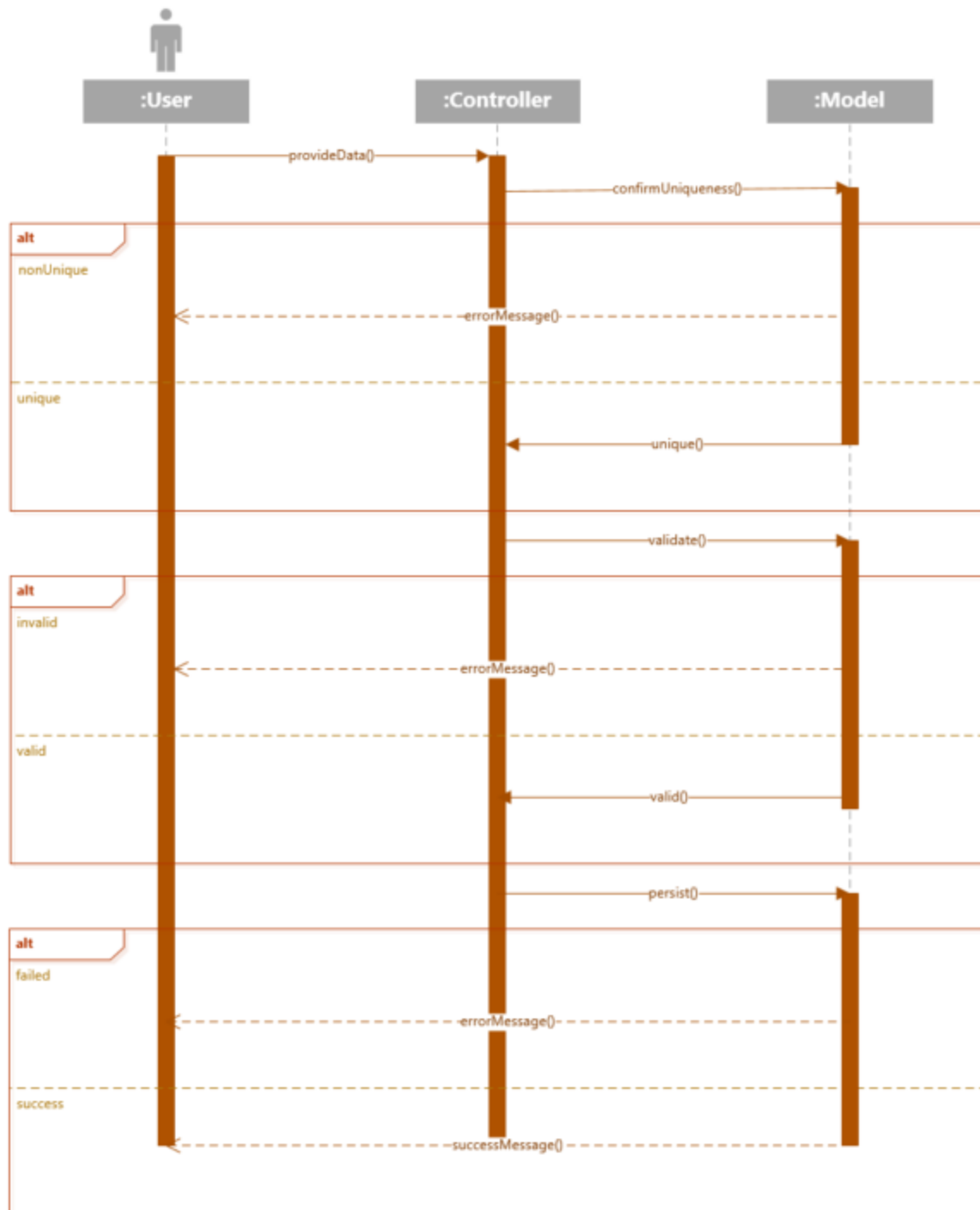
Login



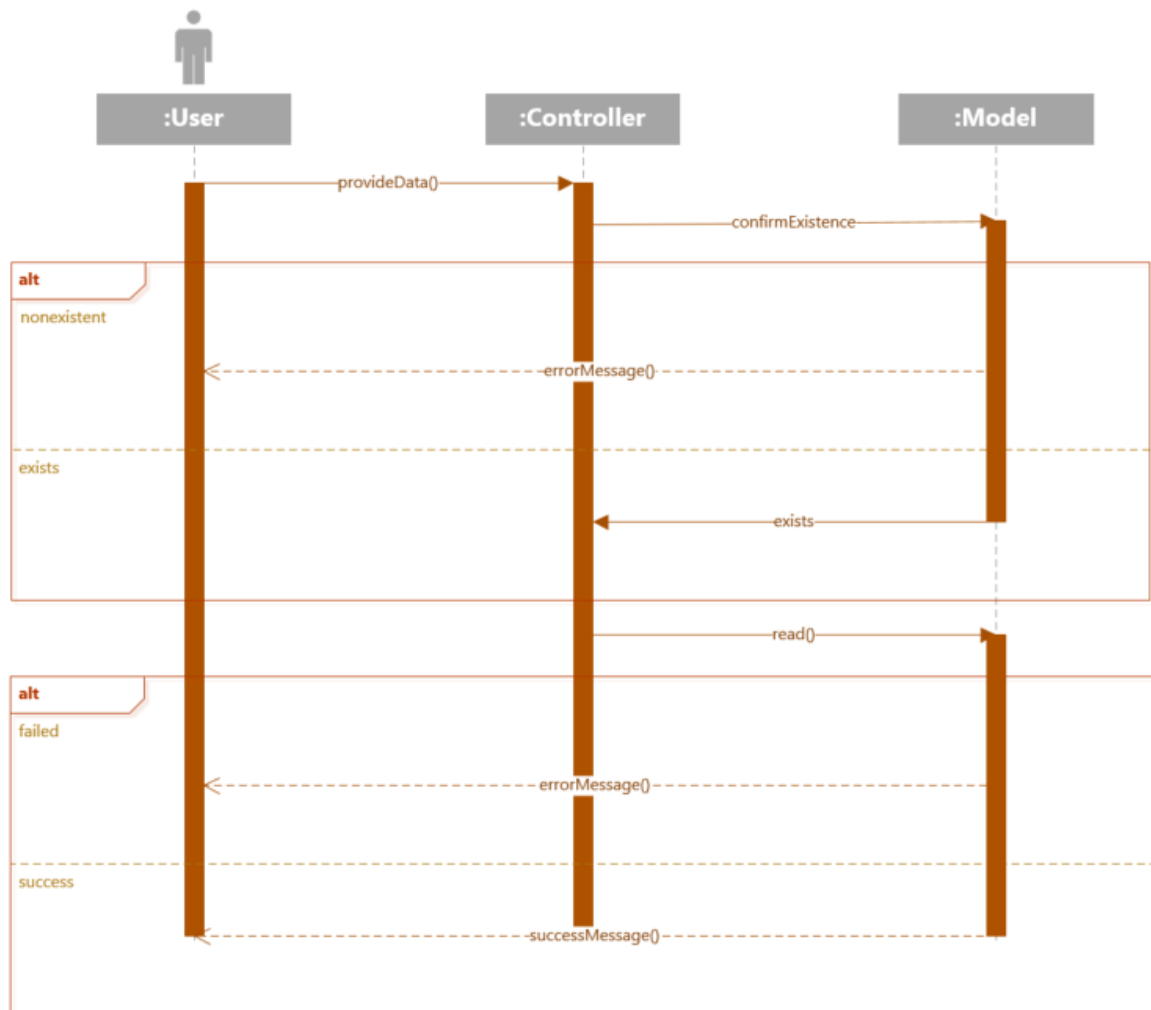
Logout



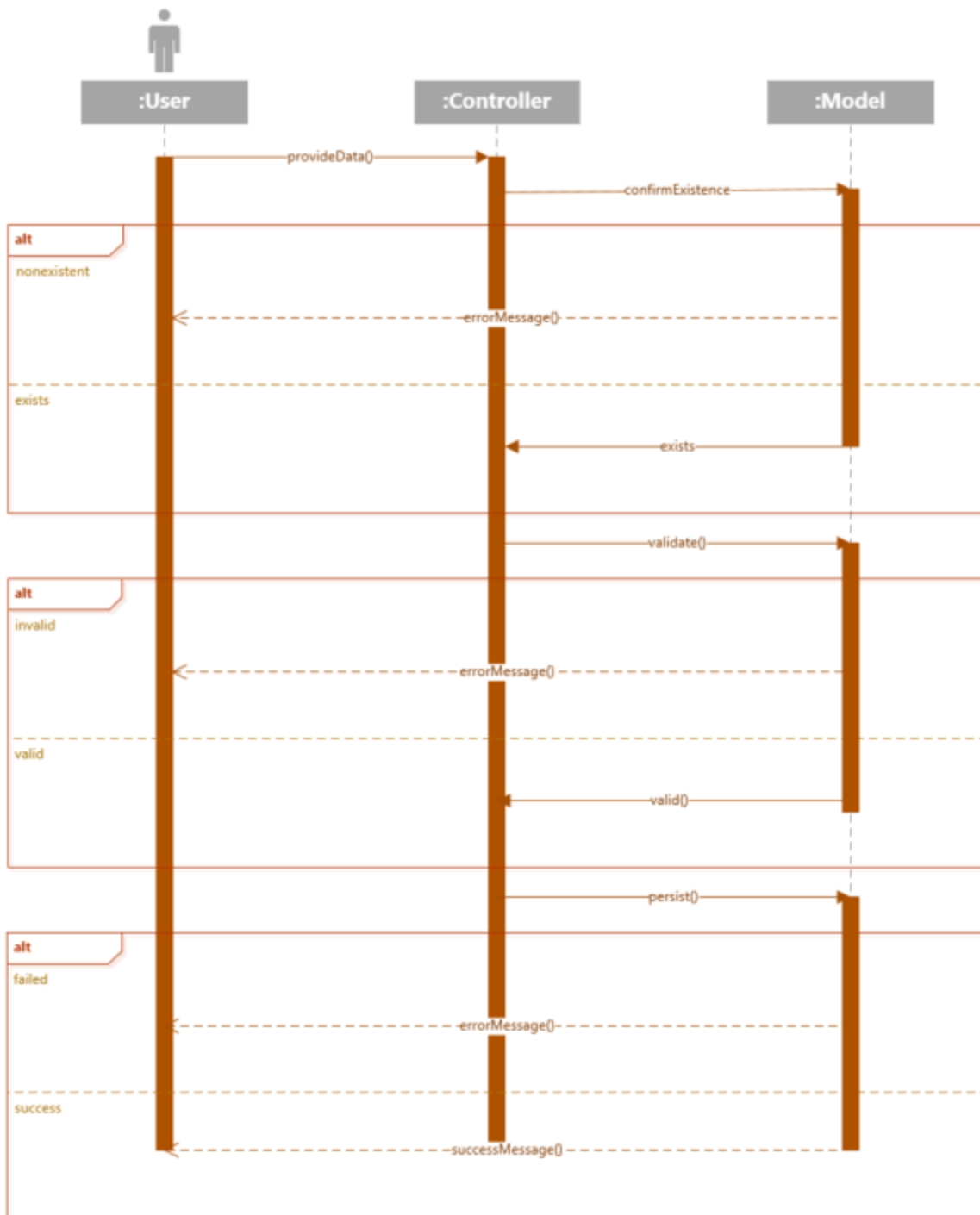
Create



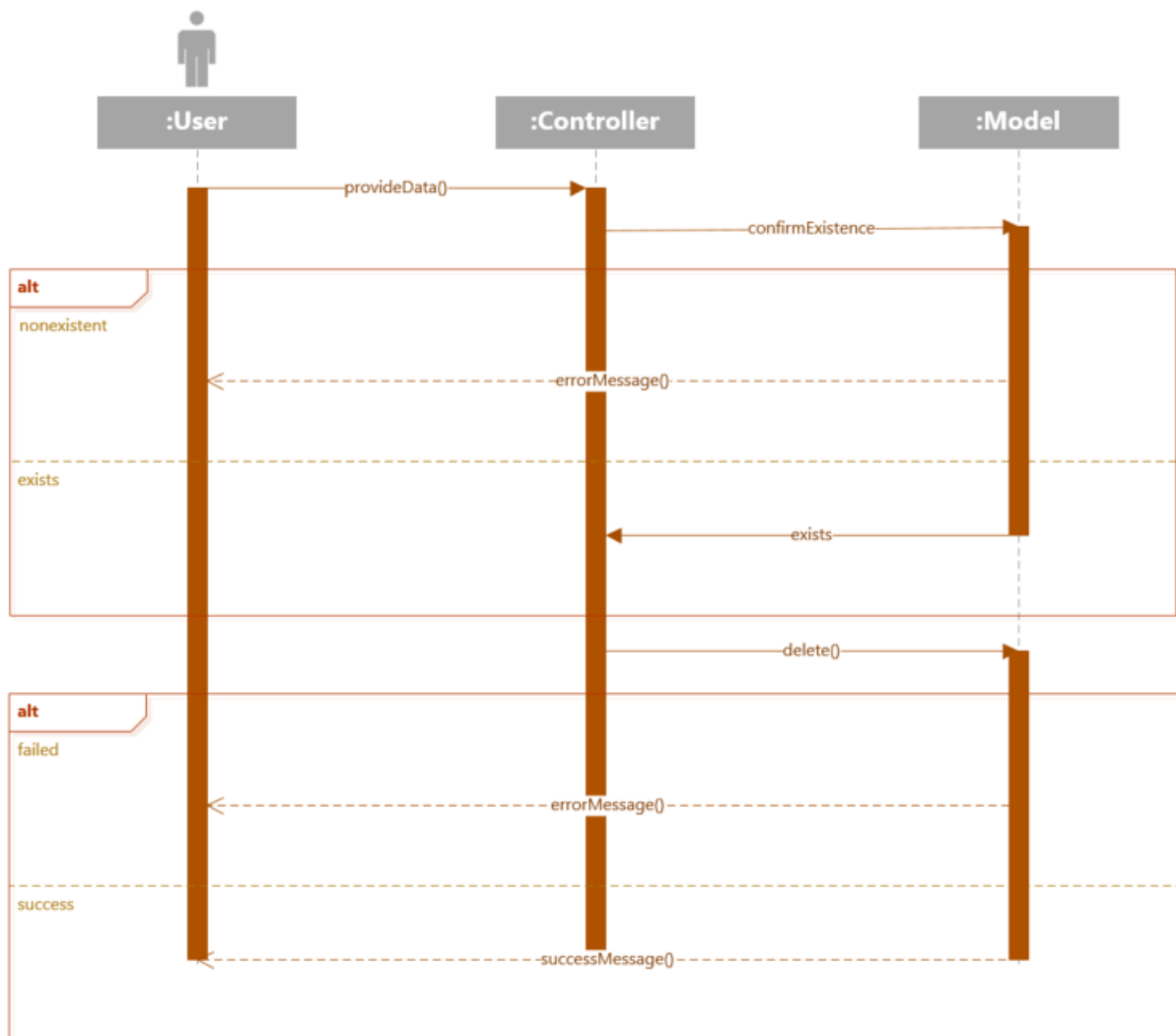
Read



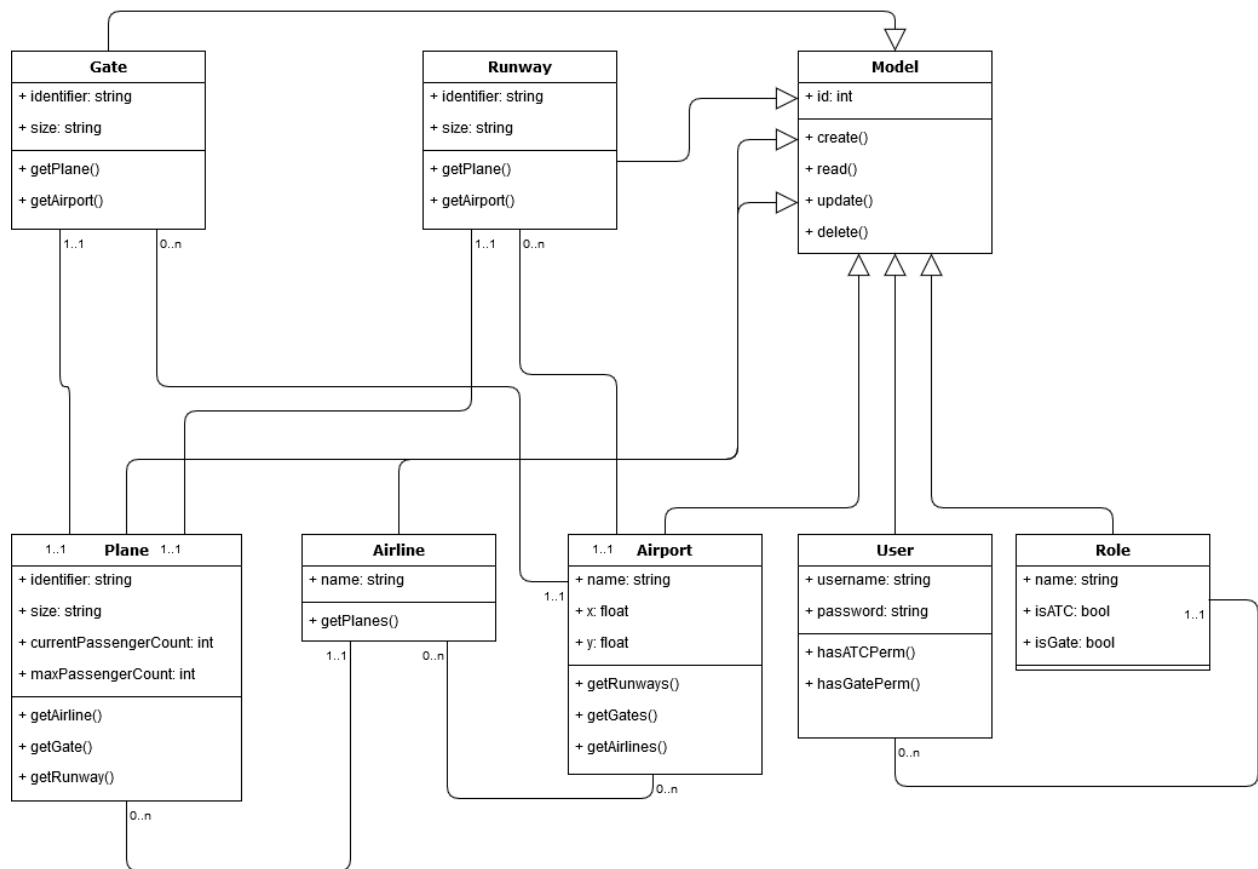
Update



Delete

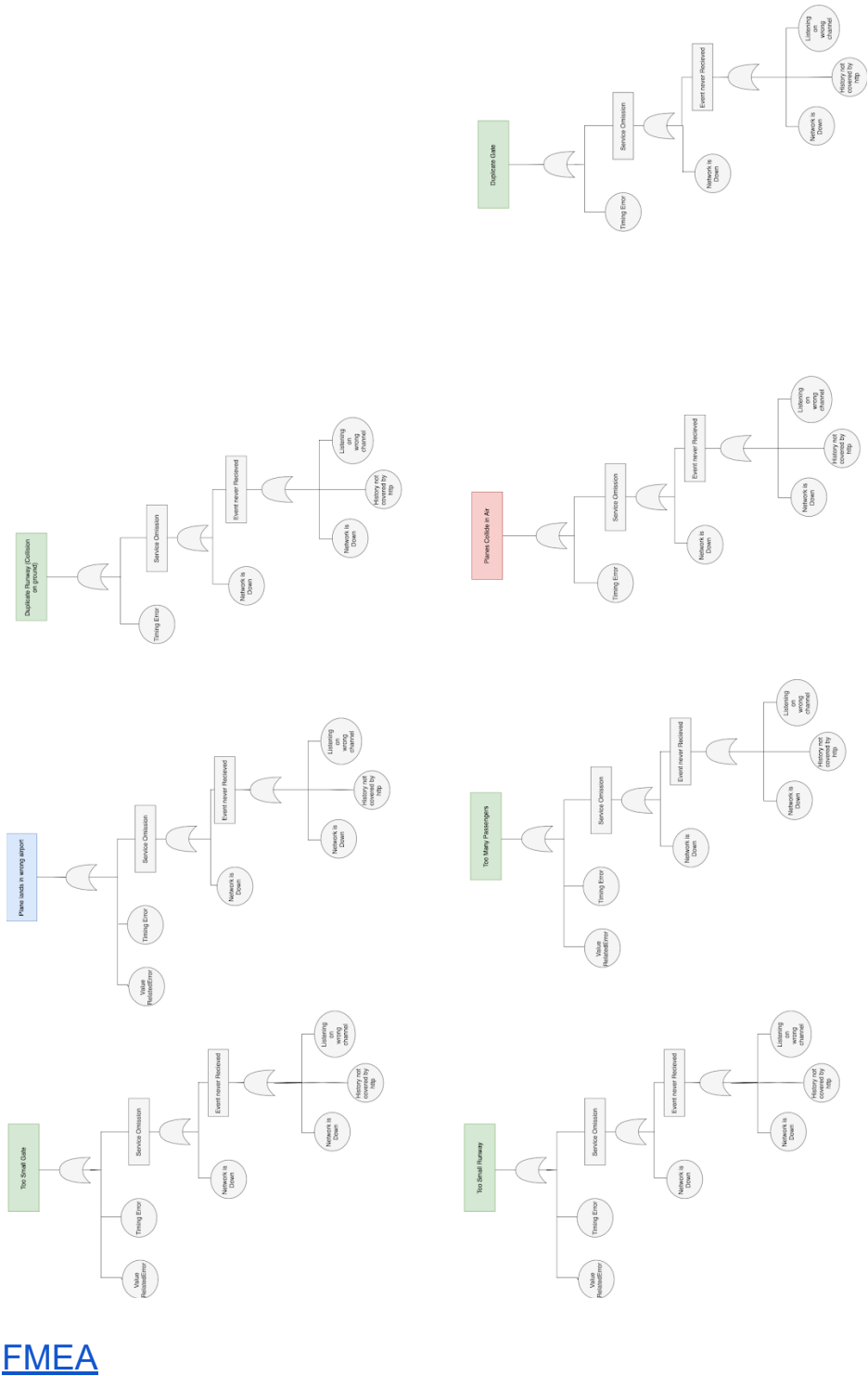


Class Diagram



Analysis Documentation

FTA



FMEA

SonarQube Analysis Results

☆ [cps420-project-drop-table-software_engineering](#)

Failed

PRIVATE

Last analysis: December 11, 2019, 7:48 PM

0   Bugs

0   Vulnerabilities

0   Code Smells

 43.9% Coverage

 0.0% Duplications

2.4k  Python, XML, ...

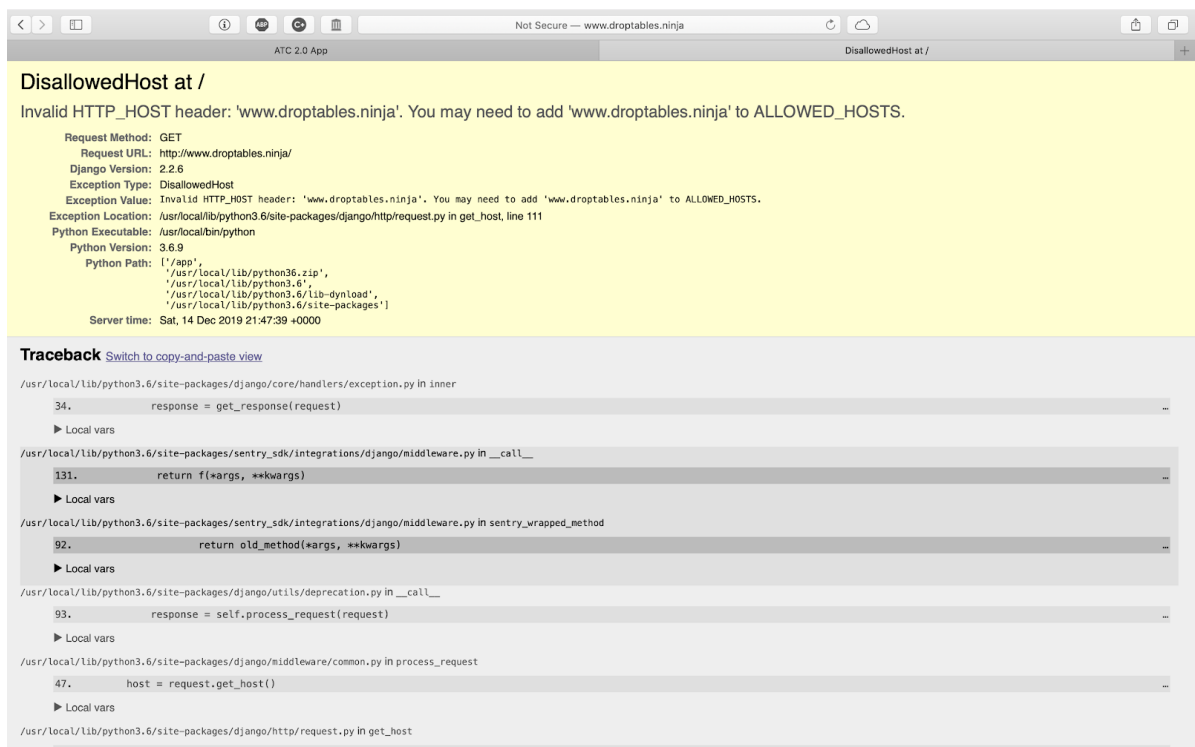
AWS Design

S3 Bucket for Static Assets

All of our static files were uploaded to an S3 bucket. This gives AWS access to them and has the added benefit of faster load times because the S3 bucket is hosted on many servers. The closest server will be the one used to load the static assets thus reducing the latency.

Route 53 Domain

Our Domain is droptables.ninja. This domain is connected to our ingress address. The settings.py file of allowed hosts must be configured properly otherwise you will receive the following error:



```
DisallowedHost at /
Invalid HTTP_HOST header: 'www.droptables.ninja'. You may need to add 'www.droptables.ninja' to ALLOWED_HOSTS.

Request Method: GET
Request URL: http://www.droptables.ninja/
Django Version: 2.2.6
Exception Type: DisallowedHost
Exception Value: Invalid HTTP_HOST header: 'www.droptables.ninja'. You may need to add 'www.droptables.ninja' to ALLOWED_HOSTS.
Exception Location: /usr/local/lib/python3.6/site-packages/django/http/request.py in get_host, line 111
Python Executable: /usr/local/bin/python
Python Version: 3.6.9
Python Path: ['/app',
              '/usr/local/lib/python3.6.zip',
              '/usr/local/lib/python3.6',
              '/usr/local/lib/python3.6/lib-dynload',
              '/usr/local/lib/python3.6/site-packages']
Server time: Sat, 14 Dec 2019 21:47:39 +0000

Traceback Switch to copy-and-paste view
/usr/local/lib/python3.6/site-packages/django/core/handlers/exception.py in inner
34.         response = get_response(request)
    ► Local vars

/usr/local/lib/python3.6/site-packages/sentry_sdk/integrations/django/middleware.py in __call__
131.         return f(*args, **kwargs)
    ► Local vars

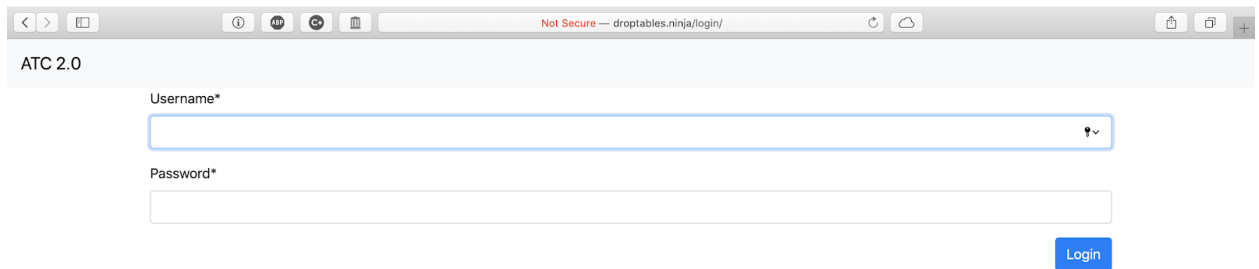
/usr/local/lib/python3.6/site-packages/sentry_sdk/integrations/django/middleware.py in sentry_wrapped_method
92.         return old_method(*args, **kwargs)
    ► Local vars

/usr/local/lib/python3.6/site-packages/django/utils/deprecation.py in __call__
93.         response = self.process_request(request)
    ► Local vars

/usr/local/lib/python3.6/site-packages/django/middleware/common.py in process_request
47.         host = request.get_host()
    ► Local vars

/usr/local/lib/python3.6/site-packages/django/http/request.py in get_host
***
***
***
```


Our application is available at <http://droptables.ninja>



ATC 2.0

Username*

Password*

Login

ECR (Elastic Container Repository)

We have an ECR (Elastic Container Repository in US-East-1 (Virginia). ECR is an AWS-provided container repository that offers version control for our docker build images. Our code is pushed here using Github and CircleCi.

VPC

Our VPC (Virtual Private Cloud) is an on-demand configurable pool of shared computing resources allocated within a public cloud environment. By using AWS, Docker, and Kubernetes we can be confident that our app will have all the environment dependencies needed to run the application. The VPC provides the network resources capable of running our docker image in our kubernetes pods.

EKS

EKS is managed Kubernetes on AWS. We are using EKS with an EC2 instance worker node to run both our development environment and our production environment. Both

are in the same cluster but in different namespaces to save on cost. We can push to either the development or production environment depending on requirements and testing.

How the Various Components of AWS Work Together

Our docker image is uploaded to our ECR. This repository holds our docker images ready to be deployed. We have manually uploaded all our static assets into an S3 bucket for faster load times. Our EKS is connected to our VPC in order to run our application. This works seamlessly because of the containerization provided by the use of Docker.

Our EKS namespaces are updated manually via command line to set the image for each namespace (development or production). For ease of access, we purchased a domain for the application (<http://droptables.com>). This Route 53 domain is connected to the elastic load balancer and pointed at Kubernetes via an ingress controller.

Code Architecture (JOANNA - CARLOS)

App Working Together (Evently, Rest, SonarQube)

The front end of the app allows the user to input information on the droptables.ninja page. The model component validates the data entered by the user (or from data sources). The view component of the app populates the page with data received from the data sources or from the user.

Evently then sends packages from the REST app to the database, updating it so it reflects the conditions for airplanes and airports is in. Following the update, the REST component monitors for errors such as duplicate entries and collisions. Should an event be thrown, Evently receives the event and manipulates the database accordingly.

Unit tests analyze all parts of the code (model, view, REST) to make sure the code is working properly while the app is offline.

The SonarQube component checks for code coverage and smells. Code coverage determines how much of the app is covered by the other resources (such as unit tests). Our standard for code coverage was 80%, which we exceeded on CircleCI (94%); however, due to a configuration issue, we could only attain a maximum of 40% on Qube.

In addition, SonarQube monitors for smells, or bugs. It applies cyclomatic complexity analysis to determine if some code is too complex. It also monitors for missing IDs and formatting issues, among others.

Code Framework and Database

The program is built on the Django Framework and uses a PostgreSQL database. It also uses a model-view-controller architecture where the database is the model, the controllers are in the view files, and the view is the Templates folder.

Database Models

The database model is made up of five objects: Airports, airlines, planes, runways, and gates.

The models are found in models.py and occupy a table in the database. The program also uses the Django built-in models for users and authorization.

See the Class Diagram figure on page 12 for the relationships between the five main models of the project.

Views

The main form of communication between the client and the database is the view files. Views that deal with specific model objects are found inside the ATC2_0 directory in view files of their model's name, the format is <model_name>_views.py, ex: plane_views.py.

Every model view is responsible for simple CRUD operations such as reading and creating new model objects.

Model views also validate input and check the permissions to perform each action.

Create

To create a new object, the user must have that object's corresponding add permission, ex: perms.ATC2_0.add_plane.

The view then compares the new object identifier or name against the existing objects of the same type in the database. If the new object shares an identifier or name with an already existing model object of the same type, the view returns an HTTP 400 error.

Otherwise, if the check passes, a new object is created in the chosen model's table with the information the user fills in the create page.

Note: names and identifiers are not the only enforced unique fields. The plane views also handle duplicate locations so as not to have two airports in one location.

Read

All users by default have Read permissions. When the user launches the app, all existing objects of one type display on the page; a user with no further permissions can simply see they exist and do nothing more.

Update

To update an existing object, the user must have Update permissions, ex: perms.ATC2_0.update_plane. On the app, the user clicks on the Edit button next to the entry and adjusts it as he wishes. Then he can click on the 'Done' button to post the changes.

Behind the scenes, the view looks up the entry selected and makes the changes posted by the user. If the user attempts to make an invalid change (such as leaving out a required field or trying to create a duplicate), the form will return a '400 Invalid Form' error and cancel the update. Otherwise, the changes are posted and then reflected to the app automatically.

Delete

To delete an existing object, the user must have Delete permissions, ex: perms.ATC2_0.delete_plane. On the app, the user clicks on the Delete button next to the entry he wishes to delete. A confirmation message will appear asking the user to confirm deletion. If the user confirms, the entry on the app is removed.

Behind the scenes, the view looks up the entry selected for deletion by the user and removes it from the database, updating accordingly.

REST Request Handling

The REST Request handlers are found in ATC2_0/views.py. The REST endpoints receive a JSON REST package and check it for errors. If an error occurs, a response REST package is sent to https://evently.bjucps.dev/app/error_report in the following format:

```
{  
  "team_id": "DROP-TABLE-SOFTWARE-ENGINEERING",  
  "obj_type": "PLANE",           // the object type which caused the error  
  "id": "AE01",                 // the object that caused the error  
  "error": "COLLISION_IMMINENT" // the error message  
}
```

The project has a total of four REST Endpoints and will throw an error when navigated to without a valid REST package.

/atc/api/headings

Receives:

```
{  
  "plane": "AE01",      // guaranteed to be an existing plane identifier  
  "direction": 0,       // guaranteed to be a value between 0 and 359  
  "speed": 0,           // guaranteed to be a positive number  
  "origin": "ATL",      // guaranteed to be an existing airport  
  "destination": "LTA" // guaranteed to be an existing airport  
  "landing_time": "2019-10-29 23:00",  
  "take_off_time": "2019-10-29 23:00",  
}
```

This view is handled in handle_heading_publish().

When a plane is created, the information is first parsed into the Plane object and converted to the appropriate type. Then, extra information is filled in according to the information in the plane object. For example, take_off_airport matches with the 'origin' member of the plane object.

Then, data checking occurs. Check_airport() verifies that the plane is at the specified airport.

Set1/2 receives all planes except those that take off and land at the same airport and has them checked with check_set1()/set2(). Then Set3 checks for planes intersecting in the sky. After all the checking is complete, an appropriate HttpResponse() is sent.

/atc/api/gates

Receives:

```
{
```

```
"plane": "AE01", // guaranteed to be an existing plane identifier
"gate": "G01", // guaranteed to be an existing gate identifier
"arrive_at_time": "2019-10-29 08:00" // optional key
}
```

This view is handled in `handle_gate_publish()`.

First, the function retrieves the identified plane and gate from the database and gate to the plane.

Next, it checks if `arrive_at_time` is in the body of the package. If it is, the function first checks the size of the plane against the size of the gate. If the gate is smaller than the plane (ex: large plane on a medium gate), the view posts a request with the error message

“TOO_SMALL_GATE.” Afterward, it loops through the gate’s plane assignments to check if any other plane assigned to this gate has landed or is landing at the same time as the plane in the request. If this condition is true, the view posts a request with the error message

“DUPLICATE_GATE.”

If `arrive_at_time` is not present in the request, the plane is considered to have finished its trip safely and its runway and arrival time are both set to null.

`/atc/api/runways`

Receives:

```
{
  "plane": "AE01", // guaranteed to be an existing plane identifier
```

```
"runway": "R01", // guaranteed to be an existing runway identifier  
"arrive_at_time": "2019-10-29 08:00"  
}
```

This view is handled in `handle_runway_publish()`.

This view assigns a plane to a runway in the database and then checks the size of the plane against the size of the runway. If the runway is smaller than the plane (ex: large plane on a medium runway), the view posts a request with the error message “TOO_SMALL_RUNWAY.”

The view then loops through the gate’s plane assignments to check if any other plane assigned to this runway is landing at the same time as the plane in the request or another plane has already landed at the runway. If this condition is true, the view posts a request with the error message “DUPLICATE_RUNWAY.”

`/atc/api/counts`

Receives:

```
{  
  "plane": "AE01", // guaranteed to be an existing plane identifier  
  "passenger_count": "R01", // guaranteed to be a positive number  
}
```

The package is handled in the function `handle_passenger_count()`.

The function retrieves the identified plane object in the request and compares the `passenger_count` key to the `max_passenger_count` variable of the plane object. If the

passenger_count is larger than the plane's max capacity, it posts an error with "TOO_MANY_PASSENGERS."

Templates

The templates are the user's connection with the views and the database. They are made of a file system of html, css, and javascript files and is build using the Bootstrap Framework.

Unit Tests

The unit tests are found in ATC2_0/tests.py

Automation Architecture

Our infrastructure was created using an ansible playbook. If needed, we can delete and recreate at any point if there is a failure. By using infrastructure as code, we minimize the risk of human error when creating resources.

When a commit is made CircleCi is configured to run multiple commands in the config.yaml file:

- The AWS account credentials are set
- CircleCi checks out the Circle branch
- CircleCi checks the requirements and installs any needed requirements on the CircleCi server
- CircleCi runs a series of tests
- CircleCi sets database variables
- CircleCi calls sonar and reports our code coverage

- CircleCi builds the docker image
- CircleCI runs the docker image in a test deployment
- CircleCi pushes docker image to our ECR

At this point the docker image is ready to be pushed to the development or production environments but is not live until we run the “set image” command in Kubernetes.

CircleCI is a CI/CD tool that checks the latest build with SonarQube for bugs and good code quality. After it passes/fails the SonarQube test, CircleCI builds the docker image from the latest code build and pushes it to ECR to be used by Docker. CircleCi is also connected to docker and SonarQube. Because these tools are connected together we can commit code and automatically have a docker image prepared, have the code tested with Sonar, and have the code uploaded to our AWS ECR.

The last step, updating our kubernetes pods with our updated code, requires user interaction as a safety measure. This is crucial during our code freeze time each week. By pushing each new docker image manually to Kubernetes, we can easily roll back to an earlier build and it also requires a person to merge code and make changes to environments.