Noyo Coding Challenge

Implementation Options

You have two options for completing this exercise:

1. Complete it prior to your interview. In the live session, we will review your solution and work through one or more of the Live Extensions

2. Complete it during your interview. If you choose this option, we ask that you still familiarize yourself with the exercise and ensure you can start the server prior to your live session.

Address Segment API

Overview

Time-based data is an important concept here at Noyo, frequently we need to understand not only what the current state of a model is, but also what it looked like at a point in the past (and sometimes the future). The purpose of this application is store records of people and to track their address over time.

The Existing Codebase

We have provided you with an existing API application built in Flask. We've gone ahead and implemented following:

- Database tables and models for both `Person` and `AddressSegment`
- API Endpoints to manage the `Person` model
- A unit test suite to test the `/person` endpoints and a subset of the `/address` endpoints

You can find instructions for running the server, the test suite, and other information in this document.

The Challenge: Create a New Endpoint

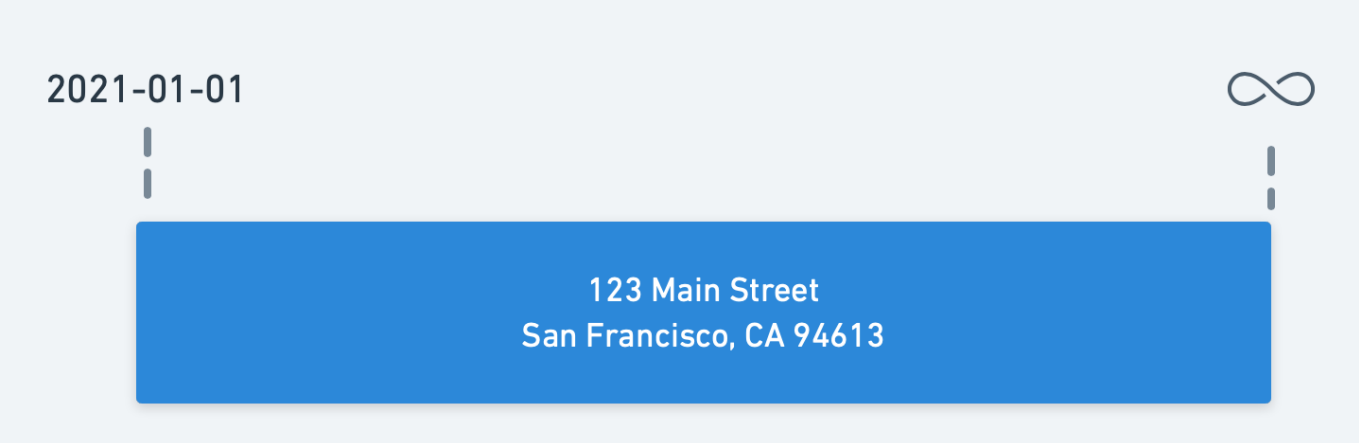
For this exercise, you'll implement the `PUT /api/persons/<person_id>/address` endpoint, which will create an `AddressSegment` for the `Person` with the specified ID.

To begin you will need to start the server and create some `Person` records either by making calls to the server or running the seed helper we've provided.

After you've created at least one `Person`, plug their `id` into the following `curl` command (or construct an API call using the tool of your choice).

```
curl -X PUT \
  http://localhost:3000/api/persons/<person_id>/address \
  -H "Content-Type: application/json" \
  -d '{
    "start_date": "2021-01-01",
    "street_one": "123 Main Street",
    "city": "San Francisco",
    "state": "CA",
    "zip_code": "94613"
  }'
```

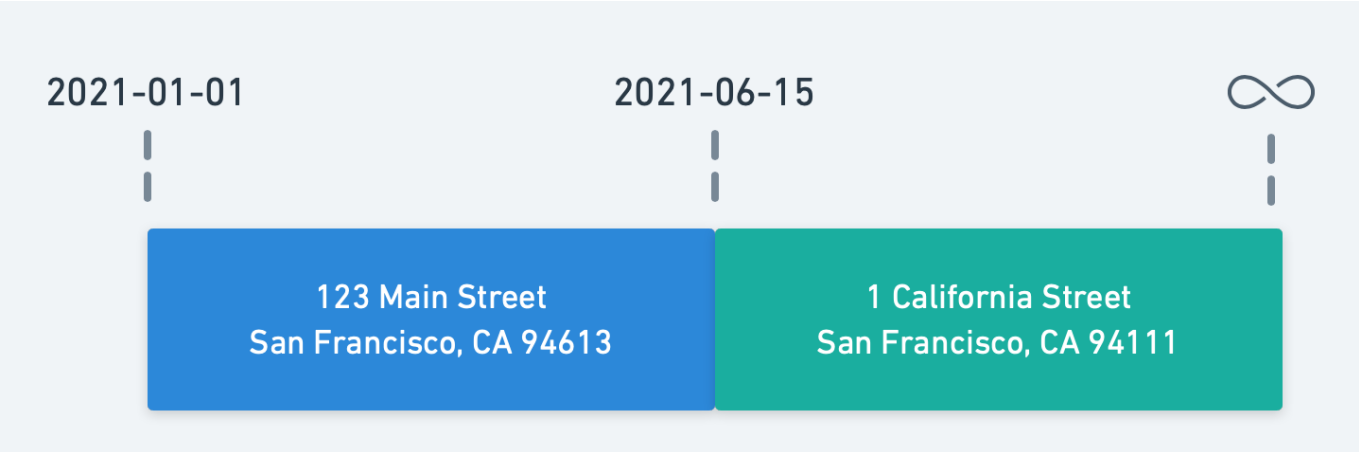
Making the previous API call will create a single `AddressSegment` record for the `Person` with a `start_date` of `2021-01-01` and an `end_date` of `null`. In our data model this means that the person's address was `123 Main Street San Francisco, CA 94613` starting on `2021-01-01` and will remain that indefinitely. The following diagram is a visual representation that:



Now we make another API call that updates the Person's address to `1 California Street San Francisco, CA 94111` starting on `2021-06-15`. **In the application's current state this should result in the server raising a `NotImplementedError`.**

```
curl -X PUT \
  http://localhost:3000/api/persons/<person_id>/address \
  -H "Content-Type: application/json" \
  -d '{
    "start_date": "2021-06-15",
    "street_one": "1 California Street",
    "city": "San Francisco",
    "state": "CA",
    "zip_code": "94111"
  }'
```

Your challenge is to update the code in this endpoint (`service/api/addresses.py`, line 68) to handle the creation of subsequent address segments. In this example, after the API call has been issued we would expect the segments of the address to look like the following diagram:



In order to assist with implementing this endpoint we have provided a failing test case (`tests/api/test_addresses.py`, line 99) that matches the example laid out above.

Live Extensions

Note: During your live coding interview, we will ask you to implement one or more of the following extensions to the code. You should not implement any of these extensions prior to the interview. You are welcome to look at them if you'd like, but we don't want you to spend any additional time implementing them beforehand.

Extension One: `start_date` Validation

Our database schema does not allow for more than one Address Segment with the same `start_date` for the same Person. If a caller sends an API request to the server with the same `start_date` as an existing Address Segment the server should return the following response with a status code of `422`:

```
{
    "error": "Address segment already exists with start_date 2021-01-01"
}
```

Your task for this extension is to implement this behavior on the `PUT /api/persons/:id/address` endpoint. In order to assist with implementing this behavior we have provided a failing test case (`tests/api/test_addresses.py`, line 121).

Extension Two: Get Address by Date

You may have noticed that the `GET /api/persons/:id/address` endpoint only returns the latest address segment. In most cases this is the desired behavior, but in some cases we may want to go back in time (or forward) to see what the address was or will be. The ability to query the address by date should be implemented as follows:

`GET /api/persons/:id/address?date=YYYY-MM-DD`

The endpoint should return the address segment that corresponds to the `date` provided in the query string. In order to assist with implementing this behavior we have provided a failing test case (`tests/api/test_addresses.py`, line 140).

Extension Three: Merge Contiguous Identical Address Segments

You may have noticed during your development that if you submit two identical address segments with different `start_date`'s that the API will create a new address segment record for each. In order to reduce the number of redundant records in the `address_segments` database, you should update the `PUT /api/persons/:id/address` endpoint to check if the new address segment is identical to the latest segment, and if it is it should skip creating a record and return the already existing segment.

In order to assist with implementing this behavior we have provided a failing test case (`tests/api/test_addresses.py`, line 164).

Instructions For Running

Here at Noyo we rely on Docker Compose to ensure portability of our applications and let developers spend more time developing and less time configuring their local environment. We've provided a `docker-compose.yml` that installs all the appropriate dependencies and creates a database that the application can connect to.

Note: Docker Compose is now bundled along with Docker for Desktop on both Mac and Windows. You may use the syntax `docker compose` rather than `docker-compose` for the following instructions.

Building the App

You can build the server and database containers using the following command:

```
docker-compose build
```

Running the App

You can start the server and database using the following command:

```
docker-compose up
```

This will start up the Python Flask server on `localhost:3000` and a PostgreSQL instance on `localhost:5432`.

Flask starts up in debug mode, meaning whenever you save a file it will restart the server with your changes.

Running the Tests

We have provided a test suite that uses pytest. You can run the test suite using the following:

```
docker-compose exec service pytest .
```

Note: To keep things simple the tests use the same database as the local server. This means when you run tests the database will be cleared of all data.

Seeding the Database

To make it easier to work with the API, we've provided a simple Python script that you can run to create 5 random `person` entries in the database. You can run the seed using the following:

```
docker-compose exec service python seed.py
```

Note: You can only run the seed after you have started the server using `docker-compose up`

Connecting to the Database

The schema for the database is created for you when you start up the container by exeuting the contents of `schema.sql`. If you would like to connect to the application's database, you can do so using the PostgreSQL client of your choice and using the following configuration:

Parameter	Value
Host	127.0.0.1
Port	5432
Database	coding_challenge
Username	noyo
Password	noyo

Formatting your Code

Here at Noyo, we use Black to handle the formatting of our code. This enables to spend less time during code reviews worrying about spacing and indents and more time focusing on the content. If you would like to format your code using Black you can run the following command:

```
docker-compose exec service black .
```

Starting Over

If you get your database or local server into a weird state you can start anew by running:

```
docker-compose down --remove-orphans --volumes
```

Then go back to Running the App and start over.