2022

Synthesizing Realistic Substitute Data for a Law Enforcement Database using a Python Library

Anthony Carrola
West Virginia University, ac00026@mix.wvu.edu

Synthesizing Realistic Substitute Data for a Law Enforcement Database using a Python Library

Anthony Carrola

Problem Report submitted to the
College of Engineering and Mineral Resources
at West Virginia University

in partial fulfillment of the requirements
for the degree of

Master of Science in
Computer Science

Matthew C. Valenti, Ph.D., Chair
Thomas Devine, Ph.D.
Roy Nutter, Ph.D.

Lane Department of Computer Science and Electrical Engineering

Morgantown, West Virginia
2022

Keywords: Fake data, anonymous data, Faker, law enforcement, database, Factory Boy

Abstract

In many databases, there is private or sensitive data that should not be accessible to any but a few individuals, such as HIPAA (Health Insurance Portability and Accountability Act) protected or LE (law enforcement) data. However, there is often a need to work with the data or change it for proper and thorough testing, especially for the developers . In some cases, the developers may be authorized to access and view the data, but it is rarely allowable for that data to be changed. Further, it is unlikely, especially on a large project, that all of the developers will have the authorization to view the data. In this case, it can be profitable to have easily creatable synthetic or 'fake' data to fill the database that mimics the real data enough to be used in all the same tests and to develop endpoints and APIs that will work with the real data. There are many possible ways to achieve this, such as shuffling the sensitive data information, or filling the sensitive data with garbled information. There are, however, drawbacks to such methods, as the data then becomes unwieldy or nonsensical to work with. Therefore, for this study, a Python library called Factory Boy, was used. Factory Boy can inherit the Django database models and then be used to generate randomized but realistic looking data, capable of mimicking all the complexities of actual database relationships and information.

Acknowledgements

This project was only accomplished with the help of many people. Without them, I would never have been able to finish this project, write this report, or achieve any of what I did. To the following people, I would like to extend my gratitude for their support, aide, and encouragement:

To Dr. Roy Nutter for helping me get the graduate research assistantship position at ICAC.

To Dr. Matthew Valenti for overseeing this report, continuously advising me and answering my countless academic questions, and leading me to a successful solution.

To Tim Ricks for suggesting and overseeing this project, and for guiding me along the way with infinite patience, even when my questions were silly.

To the office at the Crimes Against Children Unit, for welcoming me in and always making work fun.

To Dr. Thomas Devine for joining my committee, even though we had never met, and helping edit this report.

To Rahul Reddy Annareddy for helping me create a few of the factories for this project.

Table of Contents

Chapter 1: Introduction

1.1 Background

In the modern world, databases are needed in just about every application. From social media such as Twitter and Instagram, to corporations, to governments themselves, there is always data to be stored. Whether it is an image for a dating profile, census data for a government, weapons inventory for the military, or tax or financial information, the data must be kept somewhere. One can hardly use any application without the application searching for or storing data in a database.

With the ubiquity of databases, comes the need for security. The data is often sensitive or private and must be protected from individuals or organizations seeking to access the data to sell it or use it, for ad placement or phishing or something worse. For this purpose, much research has been done and is consistently being done in the field of security. The security of databases is of paramount importance. However, protecting sensitive data from unauthorized access is only part of the problem. Even if outside malicious forces can be completely stopped from accessing sensitive data, large companies must still make sure that there are not malicious actors within. Thus, to actually secure data, it is necessary to restrict the accessors to relatively few and trusted individuals.

Yet, restricting the data to a few causes problems all of its own. Oftentimes, it is essential or advantageous to release parts or variations of the data to those not necessarily authorized. This can happen in a multitude of situations, such as studies, training, or software testing and development. For example, there may be a team of software developers who are not

authorized to access the data, yet they need to have the data in order to test the software that they are developing. In such cases, it is necessary to have access to near matches of the data, or, in the case of studies, parts of the data must be anonymized (usually those parts that allow for identification) while others must be kept.

This report focuses on the case of software testing and development, in particular for a law enforcement (LE) database system and how the data are faked for security, testing, and development.

1.2 Motivation

Internet Crimes Against Children (ICAC) is a national task force dedicated to investigating, prosecuting, and aiding in cases of Internet crimes against children[1]. In order to aid investigations, there is a system available, known as the ICAC Data System or IDS, that allows investigators to network, store notes, and maintain a list of cases and cybertips, among many other uses.

To facilitate IDS, there is a complex database that contains a wide variety of tables and data types, interconnected by a network of primary keys (PKs) and foreign keys (FKs). The database contains data such as investigator and investigation information, much of which is highly sensitive, e.g., case files, investigation details, persons of interest (POI) information, etc.

At the same time, in order to create the application and the database, there is a team of developers, often augmented by graduate students at West Virginia University, who are not all authorized to access the data. Further, for proper testing and developing, the developers must

[1] https://www.icactaskforce.org/

often change or display significant parts of the data. Of course, this cannot be done with the actual data, so the data must be copied and masked or anonymized somehow.

In this study, making fake data was chosen over the other methods that mostly made use of database copying. There are a multitude of reasons that led to this decision, including:

- Increased security.
- Avoidance of complex algorithms necessary to anonymize data.
- Better test data designed to test edge cases.
- Realistic appearance.
- Dynamic data that could be created as needed and changed easily if the models changed.

Taking all of these reasons into account, research was done, and a Python library called Factory Boy[2] was chosen for its versatility and easy compatibility with the project and Django, a Python open-source web framework. Not only was the use of it fairly simple, but Factory Boy has the advantage of being able to create data 'factories' that can inherit from the Django ORMs (Object-Relational Mappers). Django ORMs are, in the simplest terms, Python objects which encapsulate the database tables and allow for easy access and manipulation of SQL without actually writing SQL code. In this way, Factory Boy allows for the fake data to change automatically, or nearly so, along with the Django ORMs.

In Chapter 2, a short literature review will be provided, surveying a research paper related to the topic. As there were not many papers directly related, only one paper will be reviewed.

[2] https://factoryboy.readthedocs.io/en/stable/

However, the chapter also includes a short description of the various methods to anonymize, mask, and fake data, since they parallel this project very closely and were considered for use.

Once the review of related research and methods is complete, Chapter 3 will introduce the methodology of this project, starting with the basics of Factory Boy and Faker, how they are related, and how they were used, before slowly moving into the complexities of each and how the libraries were leveraged to achieve a satisfactory synthesis of the sensitive data. Chapter 4 will continue the discussion by describing how the capabilities of Factory Boy were leveraged to create classes capable of generating the data simply and cleanly, usable across the system as needed. It will further describe how the factories were tested to verify that they were producing the proper data.

Finally, Chapter 5, will discuss the success of the project and the future work that can and should be done on the subject of creating fake versions of sensitive databases for testing, training, and development purposes.

Chapter 2: Literature Review

In this section, related efforts will be discussed along with their differences, weaknesses, and strengths. While there are not many reports on the synthesization of data for complex databases, multiple references were found that generated simple data in a variety of means. One such paper [1] is discussed below for its recency, but can be considered a general representative of an entire category of studies that mainly focus on specific and simpler data synthesis with the larger goal of testing, rather than necessarily making sensible data for training and demonstration as well as testing.

Another related method, discussed below, is database copying, which is the process of copying the database to create a test/demo environment, and then masking or anonymizing the data somehow. This report will provide a high-level overview, discussing the various branches of data masking and anonymization that are used after the database is copied. The literature for this method, however, strays towards unrelated subjects, such as machine learning [2], data anonymization for security [3], or the anonymization of very specific types of data, e.g., images [4] and therefore will not be pursued in depth.

2.1 Rashid Et Al

In one of the more recent studies, Rashid Et Al [1], a Genetic Algorithm is used to simulate data that works well for tests. The Genetic Algorithm works as a sort of ‘natural selection’ search, by measuring the best candidates and using them to, in essence, ‘breed’ the next generation of data. What is called a *fitness function* is used to determine the best candidates, and in the case of this study, the fitness function is designed to determine whether or not the test

data causes an error in the unit tests. Of course, the reason that an error is desired, is because it implies the data is testing the edge cases more thoroughly. For this reason, the more errors caused, the more suitable the data is for reproduction to form the next generation. The data would continue to ‘reproduce’ until the target fitness is reached, or in this case, the target number of errors.

While the research seems to have a promising future, it is still in its infancy and was solely tested on the generation of primitive data types, such as byte, short, int, etc. Further, it focuses on speed and the data’s ability to generate errors, and the algorithm's ability to create realistic looking data (as would be needed to test or mimic databases) is not clear. Likely, the algorithm’s fitness function would need to be modified significantly for this to be achieved.

2.3 Database Copying/Data Anonymization

Database copying is the method of copying real data to a test system. In some cases, when the users of the test system are completely authorized, the test data may not be anonymized in any way. However, in other cases, the test system may be open to unauthorized users such as developers or trainees that need to use and manipulate the data in various ways, and therefore the sensitive data must be hidden in some way–either expunged altogether, randomized, or anonymized. There are a variety of ways to achieve this, and given the relation to this report work, several of the most relevant methods are as follows:

- Data Encryption [5]:
    - In this method, the sensitive data is encrypted using some sort of hashing algorithm. While highly secure, it is complex to implement, and the encrypted data essentially becomes worthless to those users without a key.

- Data Scrambling [5][6]:
    - Sensitive data is scrambled in random order. In this case, an ID or SSN might be randomly reorganized to be illegible. While simpler than data encryption, it is also far less secure.
- Data Shuffling [5][6]:
    - The data is shuffled so that the ID for one John Doe may be given to Jane Doe, or that the phone number for one user is actually assigned to another user. While this might be useful in some situations, plenty of sensitive data will clearly still be available, such as credit card numbers, SSNs, phone numbers, addresses, and more.
- Data Substitution [5]:
    - This method is to replace the sensitive data with realistic looking fake data. An example would be to replace a user's SSN with a randomly generated number of the same length and appearance. This method is safe and has the advantage of hiding the sensitive data while still making the data seem realistic. The difficulty, of course, is how to create the fake data and the algorithms necessary to replace the sensitive data. Further, if significant amounts of data must be faked, as in the case of this study, then the question becomes if any real data should be used.

Chapter 3: Factory Boy

3.1 An Introduction to Factory Boy and Faker

Factory Boy[3] is a Python library that was designed to create an easy way to generate fake data using Python, with a focus on object-oriented design, so that an entire object may be easily faked without duplicating all of the code necessary to define the object in the first place.

A 'factory' is the term that is used to refer to the model that inherits a specific already existing model and then creates a version of it with the fake data described in the class. In simpler terms, it is a model that overrides an existing model to create a 'fake' version.

```
import factory
From . import student
class StudentFactory(factory.Factory):
      class Meta:
            model = student
     first_name = factory.Faker('name')
```

Figure 1: An Example of simple object Factory

Figure 1 gives an example of this, showing a very basic sample factory for an arbitrary model called *student* that has a single required attribute of *first_name*.

The Meta class in Python defines the type of object that is created. In Python objects, if a Meta class is not defined, then the class is based on a default Meta class called *type*. *Type* is used to define almost all objects. If a Meta class is included, then the object is now based on that Meta

[3] https://factoryboy.readthedocs.io/en/stable/

class. The point of this digression is simply to point out, then, that in the case of Figure 1, and all of the factory definitions provided throughout this report, we write

class Meta:

model = student

By doing this, we are simply telling the Meta class what type of Django ORM to create and where to save it.

Underneath the Meta class, in the body of the StudentFactory, *first_name* represents the *first_name* attribute that must exist in the *student* class. In this case, the *StudentFactory* calls a library that Factory Boy inherits heavily from called *Faker* to fake a *name*. For this model, a full name (first and last name) will be created and assigned to *first_name.* This will obviously create an unrealistic *first_name*, but is unimportant for the purpose of this example.

```
>>> Faker.seed(0)
>>> for _ in range(5):
...     fake.ssn()
...
'865-50-6891'
'042-34-8377'
'498-52-4970'
'489-46-9559'
'224-65-2282'
```

Figure 2: Faker Example

Faker,[4] as mentioned above, is another Python library, which was also considered for use in this study, but was rejected in favor of Factory Boy, as Factory Boy contains all of the attributes of Faker and further includes functionality that would have been necessary to create manually. Faker's main use is the generation of 'fake' but realistic looking data through the use of what are called *providers*. A *provider* quite literally provides random data for a given data type, such as *name* giving a first and last name, and *first_name* giving a first name, and *ssn* providing a social

[4] https://faker.readthedocs.io/en/master/#

security number. Figure 2 provides a good example of this, showing the leveraging of Faker to produce fake social security numbers.

There are a multitude of providers documented on the Faker website, providing everything from random credit card numbers and dates to random phone numbers and email addresses, to random hash values. Further, there are ways to specify which location the data should be from, as in, how should the phone number or name appear. British phone numbers are different from American phone numbers, after all, and such differences in data can be found around the world, in everything from names to addresses. In addition to these features, there is also the ability to make custom providers, if deemed necessary, which will be discussed in more detail later in this report. All of Faker's features are available within Factory Boy, as seen in Figure 1, by a simple call within the *factory* object, where 'name' is the provider desired:

factory.Faker('name')

To return to Factory Boy, there are further features worth mentioning before moving onto a discussion of how it was used to implement the project. Most of the following features will be defined in more detail by examples later, but for now a short listing of features is as follows:

- SubFactory
    - factory.SubFactory(ArbitraryFactory)
    - This feature of Factory Boy allows for easy generation of other objects within the factory, so that one-to-one fields may easily be filled. As long as the proper factory is provided, a model will be created and linked to the assigned field.
- Iterator
    - factory.Iterator(models)

    - Iterator allows for the factory to choose from already created models to be used for the one-to-one fields. However, if there are no objects already generated, an iterator error will be caused. Thus, the developers must be certain that at least one model is in existence.

- Sequence
    - factory.Sequence()
    - This feature is especially useful in the generation of fields that must be unique and therefore the typical Faker providers are not sufficient (as they may repeat occasionally, such as in the cases of usernames, email addresses, telephone numbers, etc.). The Sequence is a counter that is always unique across the multiple factories, since it is a static part of the Factory Boy class.

3.2 Factory Boy with Django ORMs

As stated previously, Factory Boy works effortlessly with Django ORMs, and is able to encapsulate them almost the same as any other object. The only difference is in the ‘factory’ that the model inherits: Instead of inheriting the general *Factory*, the factory must inherit the *DjangoModelFactory*. See Figure 3 for an example of a factory for the simple Django ORM in IDS called *Supervisor* which encapsulates the table for supervisors. This is obviously simple

enough, and Factory Boy works equally as well for most models, which clearly demonstrates why Factory Boy was chosen to create the test data for the IDS.

```
import factory
from models import Supervisor
from factory.django import DjangoModelFactory
from factories.telephone import TelephoneFactory

class SupervisorFactory(DjangoModelFactory):
    class Meta:
        model = Supervisor
    first_name = factory.Faker("first_name")
    last_name = factory.Faker("last_name")
    title = factory.Sequence(lambda n: 'title%d' % n)

    office_phone = factory.SubFactory(TelephoneFactory)
```

Figure 3: An example of a factory for a Django ORM

```
class Supervisor(CreatedUpdatedUUIDModel):
    first_name = models.CharField(max_length=32, blank=True)
    last_name = models.CharField(max_length=32, blank=False)
    title = models.CharField(max_length=32, blank=True)
    office_phone = models.ForeignKey(
        'core.Telephone', on_delete=models.SET_NULL, blank=True, null=True
    )
```

Figure 4: A simple example of a Django ORM, and the model that Figure 3 is based

Also visible in Figure 3, are examples of *factory.Faker* being used to create a first name and a last name, described in the previous section, as well as the SubFactory that was discussed above. In this case, the factory is using a *TelephoneFactory* to fill the *office_phone* attribute since the *office_phone* attribute in the original model, shown in Figure 4, requires a Telephone model. Finally, *title* displays a good example of a *factory.Sequence* at work with the word ‘title’

appearing in the text appended to increasing numbers. See Figure 5 for how this will appear in the data itself when the factory is used.

Comparing Figures 3 and 4 provides a good example of how a simple model or ORM might be modeled by a factory, and as can be seen, creating a factory for an ORM is typically a very simple and straightforward process. The attributes that are marked as *required* must be defined in the factory, or an error will be thrown. In this case, none of the attributes are necessarily required, and the names (*first_name* and *last_name)* will be automatically set to blank if not defined by the Factory.

Of course, Figure 3 shows that factory defines *first_name* and *last_name* to *factory.Faker(first_name)* and *factory.Faker(last_name)* respectively, thus ensuring that Faker will choose realistic or semi-realistic names for all Supervisor models created by the factory.

```
for _ in range(5):
    SupervisorFactory()

for sup in Supervisor.objects.all():
    print(f"Name: {sup.first_name} {sup.last_name}")
    print(f"\t title: {sup.title}")
    print(f"\t office_phone: {sup.office_phone}")
```

Output of five supervisors:

```
Name: Sabrina Wood
        title: title0
        office_phone: Telephone object (b75e446f-285c-404f-a907-cb79a1ad9d4a)
Name: Tiffany Le
        title: title1
        office_phone: Telephone object (f2845650-221c-4700-bfb1-0a95954145e4)
Name: Brittany Long
        title: title2
        office_phone: Telephone object (a4267534-d6ad-4ca1-a600-ac43ea6672dc)
Name: Renee Maxwell
        title: title3
        office_phone: Telephone object (844aab09-049c-40cf-923e-19ee76af1b43)
Name: Kimberly Peterson
        title: title4
        office_phone: Telephone object (12c8f687-8337-4c4f-84c7-df6cb12154b2)
```

Figure 5: Code and output for creating and printing Supervisors using SupervisorFactory()

Figure 5 shows an example of the creation of five randomized supervisors. Notice how *office_phone* is printed out as a Telephone object, since *SupervisorFactory()* creates one for that attribute.

3.3 More Complex ORM Models

So far, the report has shown only how relatively simple ORMs were created. This section will focus on more difficult ORMs and attributes within them. While, for the most part, model and ORM model factories are easy to produce, there are cases that are not quite so

straightforward, with models and attributes that are fairly complex. This section will also cover ways to set attributes in factories that have not yet been mentioned.

One type of attribute that has not yet been discussed, is an attribute that is defined by the other attributes. Factories do not immediately set the values of the attributes, but instead insert a promise of the attribute to be set. For instance, as shown in Figure 6, if we try to set a variable with *factory* and immediately print it out, we will get an object of factory type, instead of the variable, even for something as relatively simple as setting the name. It is, in essence, creating a promise, to set a name at a future time (i.e., when the factory is created).

```
>>> import factory
>>> name = factory.Faker('name')
>>> print(name)
<factory.faker.Faker object at 0x7f978db31070>
>>> print(name.__dict__)
{'provider': 'name', '_creation_counter': 0, '_defaults': {'locale': None}}
```

Figure 6: Showing how faker.Factory() assigns values to be defined later.

For this reason, if we want to assign an attribute to be defined by other attributes, the solution is not as simple as writing:

email = first_name+last_name+ "@" + "example.com"

The above code will not always work since the attributes are not defined until the factory is created. Instead, we should use the safer method, or what is known as the lazy attribute, which will define the attributes each time the factory is generated. Figure 7 shows an example of where

this was used in the IDS code. Similar to most of the above code, it is fairly simple to use.

```
import factory
from core.models import Account
from factory.django import DjangoModelFactory
from test_data.factories.core.associable import TelephoneFactory

class AccountFactory(DjangoModelFactory):
    class Meta:
        model = Account

    email = factory.LazyAttribute(
        lambda o: '%s%s%s@example.com' % (o.first_name,
        o.middle_initial,
        o.last_name))
    first_name = factory.Faker("first_name")
    last_name = factory.Faker("last_name")
    middle_initial = factory.Faker("random_uppercase_letter")
```

Figure 7: Part of the AccountFactory which shows an example of Lazy Attribute

Another problem that has yet to be described in this report is the need for many-to-many fields. One-to-one relationships have been demonstrated through the use of

factory.SubFactory or *factory.Iterator(objects)*,

but these methods would not work for many-to-many relationships, which requires that multiple models are associated with the single model. The solutions to one-to-one relationships are fairly elegant and easy to implement, but there is no equally easy way to achieve the many-to-many relationship. The method used in this project was the *post_generation* command, demonstrated in Figure 8. While not entirely straightforward, it is slightly less complex than the next method (shown in Figure 9), which requires the creation of an extra model and factory. The *post_generation* method requires that the models be added *after* generation, as the title implies.

Once the model is generated, the models can be added to its many-to-many relationships. This will be further discussed in a later section, where data generation is shown.

```
class ProfileFactory(DjangoModelFactory):

    class Meta:
        model = Profile

    title = factory.Faker("job")
    department = factory.Faker("company")
    account = factory.SubFactory(AccountFactory)
    office_phone = factory.SubFactory(TelephoneFactory)
    supervisor = factory.SubFactory(SupervisorFactory)

    @factory.post_generation
    def roles(self, create, extracted, **kwargs):
        if not create:
            return
        if extracted:
            for role in extracted:
                self.roles.add(role)
```

Figure 8: In this figure the *post_generation* hook can be seen, which allows for the many-to-many field, roles, to be added manually when the model is generated as will be seen in a later section.

The other method shown in the documentation is the *through* method. To use the *through* method to create many-to-many relationships, extra models and factories must be created. That is, to create a model with one related model (in a many-to-many field), there must be a specific factory, and to create a model with two related models, there must be another specific factory. This continues upwards for the various number of related models desired. Further, along with the extra factories that must be created, there must also be an extra model created as a 'link' between

them. Figure 9 outlines this behavior, showing the example on the Factory Boy website[5] that demonstrates the linking of two simple factories using the *through* method.

```python
# models.py
class User(models.Model):
    name = models.CharField()

class Group(models.Model):
    name = models.CharField()
    members =
models.ManyToManyField(User,
through='GroupLevel')

class GroupLevel(models.Model):
    user = models.ForeignKey(User)
    group = models.ForeignKey(Group)
    rank = models.IntegerField()

# factories.py
class UserFactory(
factory.django.DjangoModelFactory):
    class Meta:
        model = models.User

    name = "John Doe"

class GroupFactory(
factory.django.DjangoModelFactory):
    class Meta:
        model = models.Group

    name = "Admins"
```

```python
class GroupLevelFactory(
factory.django.DjangoModelFactory):
    class Meta:
        model = models.GroupLevel

    user =
factory.SubFactory(UserFactory)
    group =
factory.SubFactory(GroupFactory)
    rank = 1

class
UserWithGroupFactory(UserFactory):
    membership =
factory.RelatedFactory(
        GroupLevelFactory,
        factory_related_name='user',
    )

class
UserWith2GroupsFactory(UserFactory):
    membership1 =
factory.RelatedFactory(
        GroupLevelFactory,
        factory_related_name='user',
        group__name='Group1',
    )
    membership2 =
factory.RelatedFactory(
        GroupLevelFactory,
        factory_related_name='user',
        group__name='Group2',
    )
```

Figure 9: Using the *through* method to make many-to-many relationships

[5] https://factoryboy.readthedocs.io/en/stable/recipes.html?highlight=many-to-many#many-to-many-relation-with-a-through

While the *through* method requires less processing in the data generation, the downsides are significant, as it requires an excessive number of factories to be created, and even then the generation of related objects is not random. Further, there is less control over how the many-to-many relationships are set up. For these reasons, the *post_generation* option was chosen. How it was implemented will be further shown in Section 4.2.

As the final part of this section, abstract and inherited classes should be discussed. It can often be the case that an ORM or model may inherit significantly from other models, so it is important to explore how Factory Boy can model this, without every attribute being repeated. This task proved easy and intuitive enough, as one factory could inherit from another factory, and all of the inherited attributes would carry over. Figure 10 shows how it is done with two simple ORMs.

```
import factory
from core.models import Investigator
from factory.django import DjangoModelFactory

class InvestigatorFactory(DjangoModelFactory):
    class Meta:
        model = Investigator


import factory
from core.models import NCMECInvestigator
from test_data.factories.core import NCMECAgencyFactory, TelephoneFactory
from .investigator import InvestigatorFactory

class NCMECInvestigatorFactory(InvestigatorFactory):
    class Meta:
        model = NCMECInvestigator

    name = factory.Faker('name')
    email = factory.Sequence(lambda n: 'investigator%d@icacdatasystem.org'
            % n)
    agency = factory.SubFactory(NCMECAgencyFactory)
    office_phone = factory.SubFactory(TelephoneFactory)
```

Figure 10: A demonstration of how one factory can inherit another, with NCMECInvestigator inheriting from InvestigatorFactory

3.4 Custom Providers

As mentioned above multiple times, ‘providers’ are used in almost every factory to facilitate the creation of the fake data and they form the basis of Faker, upon which part of Factory Boy is built. When a *first_name* attribute is generated, it is a provider that is called through the *factory.Faker(‘first_name’)*. The provider for it would be called *‘first_name’* and if it were being called directly through Faker, the command would be written as *faker.first_name.*

There are a multitude of providers across a vast variety of different classes, categories, nations, languages, and fields, but of course, at some point in a project the size of IDS, the prebuilt providers will not be sufficient. This may be because they are not specific enough, or because something unique to the system must be synthesized. In this project, both cases were dealt with, as will be demonstrated below. It should be noted that the documentation for adding custom providers to Faker is far more complete than the different process of adding providers to Factory Boy, and it is the latter which is explored in this section.

```
import factory
from faker.providers import BaseProvider
from core.models import Account
import random

status_choices = [Account.ENABLED, Account.PASSWORD_FAILURE,
Account.ADMINISTRATIVELY_DISABLED]
class AccountStatusProvider(BaseProvider):
    def account_status(self):
        return random.choice(status_choices)

factory.Faker.add_provider(AccountStatusProvider)
```

Figure 11: A provider for Account status

Figure 11 shows the simple provider that was created to generate statuses for the Account model, which was a provider that obviously could not exist outside of the IDS system. *BaseProvider* is imported from *faker.providers* as a first step, so that it can be inherited by the following class. Once the class has been named, the actual provider to be called is defined as a function within the class. Note that multiple providers could be created in the same provider class as is actually done in a later provider. *status_choices* is defined at the top, imported from the Account model. The *account_status* provider then returns a choice from *status_choices*.

Finally, at the bottom of the file, the provider is added to Factory Boy's Faker class, so that when called in another file, it can function as a regular provider, as shown in Figure 12.

```
class AccountFactory(DjangoModelFactory):
    class Meta:
        model = Account

    first_name = factory.Faker("first_name")
    last_name = factory.Faker("last_name")
    middle_initial = factory.Faker("random_uppercase_letter")
    account_status = factory.Faker("account_status")
```

Figure 12: Demonostrating that *account_status* can now be called like a regular provider

Another instance where a provider was needed, was in the synthesizing of file hashes. A file hash is when a file is hashed by an algorithm to form a certain number. While Faker has a variety of file hash types, such as MD5 and SHA1, the IDS project deals heavily in file hashes, needing hash types that are lesser known or slightly different from the basic sort. In order to achieve the appropriate random hash types, it was necessary to create two hash providers: CRC and SHA1 Base 32. Figure 13 demonstrates how a CRC hash was generated randomly and stored in a provider for later use.

```
import random
import factory
from faker.providers import BaseProvider

class CRCProvider(BaseProvider):
    def crc(self):
        population = 'ABCDEFabcdef' + '0123456789'
        return ''.join(random.choices(population, k=8))

factory.Faker.add_provider(CRCProvider)
```

Figure 13: CRC Provider

3.5 List of Attribute Setters Used

This project used a variety of attribute setters and Fakers from the Factory Boy and Faker libraries, and as a summary, a list of them is provided below. The list is not necessarily exhaustive of the tools used in setting attributes and is certainly not exhaustive of the multitude of setters available, but nonetheless will be helpful.

- Providers
    - Both custom and default providers were used, setting everything from names and addresses, to hashes and statuses.
- SubFactory
    - The SubFactory function was used to set most one-to-one relationships, except where it was necessary that the model be one already existing in the database.
- Iterator
    - Iterators were used wherever it was necessary to have the one-to-one relationships set with models already created.
- Sequence
    - Sequence attribute was one of the most useful, in that it was used for any field that was necessarily unique.
- LazyAttribute
    - The Lazy Attribute was at first used to form email addresses, but was later replaced with the Sequence, as, occasionally, when mass data was created, the names would repeat and therefore the emails would be the same.
- post_generation

- This was used to create many-to-many relationships in data generation. Unfortunately, it was not capable of generating randomized many-to-many relationships on its own.

Chapter 4: Creating and Testing the Data

4.1 Overriding the Factories

So far, this report has shown only the most basic of the factory use, as in calling the factory like:

AccountFactory()

However, Factory Boy is very flexible in this area, so that when the factories are called the default values which would normally be randomized, can be overloaded. In this way, if a specific type of model is desired, it is exceedingly easy to produce using the base factory. For example, say that a fake account with a specific name was desired for some reason. The code could simply be written as:

AccountFactory(first_name = 'John', last_name = 'Doe')

Testing the database would reveal that an account had been created with the first name 'John' and the last name 'Doe,' and that all of the other attributes were still synthesized as the factory specified. When multiple models containing specific values are desired, storing the values in a list and then creating the models from the list worked very well, while minimizing the code. Relatively specific and complex overrides were possible while keeping the code clean and short, and allowing for easy customization, should the need arise.

In this section, it is also important to discuss something that was mentioned above but never fully described. That is, using the *post_generation* tag to add one-to-many or many-to-many related models. In order to do this, the needed models must first be created. If two Profile

models need to be related to one Account model, then we would create the Profile models and store them in a list. Then, we could create an Account model and write something like,

account = AccountFactory(profiles=profiles),

where profiles is a list of profiles. Figure 15 illustrates this point in full, showing how the process was randomized in model generation, so that a random number of permissions and roles might be assigned to a single account.

```
# get count of permissions, and then a random number of them
count = Permission.objects.all().count()-1
numOf = random.randint(1, 3)

# get permissions to add to profile
perms = []
for x in range(numOf):
    num = random.randint(0, count)
    perm = Permission.objects.all()[num]
    perms.append(perm)

# get count of roles, and then a random number of them
count = Role.objects.all().count()-1
numOf = random.randint(1, 4)

# get roles to add to profile
ros = []
for x in range(numOf):
    num = random.randint(0, count)
    ro = Role.objects.all()[num]
    ros.append(ro)

# add permission and roles and create profile
prof = ProfileFactory.create(permissions=perms, roles=ros)
```

Figure 15: An example of how *post_generation* is used to add a random number of roles and permissions to a profile

4.2 Generating the Data on a Mass Scale

Once all the factories were created, the problem still remained of how to use the factories to generate the data. For this, two different goals existed: To create data of a certain specification for a test database, and to be able to create random data as necessary for any other needs.

To achieve these goals, it was briefly considered that the generators might be included as part of the Factory classes themselves. This would have made for a simple design and easy maintenance, since the code for each model would have been in the same place. However, somewhat paradoxically, it would have spread the code for generation through a multitude of files, as there are many different factories. Each model's generator would have been separate from those related to it, as in the case of Profile and Account which are highly related models. Further, the generation of the data would have been somewhat more laborious, as each factory model would have to be called for the generation.

For these reasons, it was decided to create a generator class to be called for the generation of data. With the amount of models in the database, however, one generator class would have become a massive file, so the data generation was split up into multiple generator classes, one for each subset of models, based on how they were partitioned in the original code (e.g., Profile and Account are part of the *user* folder, and are therefore generated in the UserGenerator).

Each generator then contains a list of functions to generate specific and generalized types of data. For instance, the UserGenerator has functions to generate both Profile Models and Account Models. Those functions are further split up into functions that generate randomized models and functions that generate prescribed models for the test database being designed, for optimal flexibility.

```
def generate_accounts_and_profiles(self):

        icac_agency = Agency.objects.get(name="ICAC Data System")
        if not icac_agency:
            print("Error. Agencies need created")
            print("Creating...")
            AgencyDataGenerator.generate_agencies()
        admin_account = AccountFactory(
            email="admin@icacdatasystem.com")
        admin_account.set_password("testpass")
        admin_profile = ProfileFactory(
            account=admin_account, agency=icac_agency)

        # for every agency, call generate_accounts_profiles_loop
        # which recursively tracks through the agencies and creates profiles and accounts
        for agency in icac_agency.children.all():
            self.generate_accounts_profiles_loop(agency)

        agency_string_list = [
            {"cur_ag": "US Southwest BLE", "sec_ag": "US Northwest BLE"},
            {"cur_ag": "US Northwest BLE", "sec_ag": "US Southwest BLE"},
            {"cur_ag": "US Northeast BLE", "sec_ag": "US Southeast BLE"},
            {"cur_ag": "US Southeast BLE", "sec_ag": "US Northeast BLE"}
        ]

        for new_ag in agency_string_list:
            cur_agency = Agency.objects.get(name=new_ag['cur_ag'])
            sec_agency = Agency.objects.get(name=new_ag['sec_ag'])
            profile = cur_agency.profiles.first()
            account = profile.account
            new_profile = ProfileFactory(
                account=account, agency=sec_agency)
```

Figure 16: An example of a function in the UserGenerator class, which generates specific accounts and profiles of agencies already created.

Figure 16 shows an example of a function in the UserGenerator class, showing how specific Profiles and Accounts are made for the test database. While the whole class is too large to show here, there are multiple other functions. For comparison, Figure 17 shows a random Account function in the class.

```
def generate_random_accounts(self, num):
        try:
            num = int(num)
            for _ in range(num):
                acc = AccountFactory()
        except ValueError:
            raise Exception("Number must be an integer")
```

Figure 17: An example of a random generator function

As a final note, part of the way through, the design for the random generators were changed from hardcoding a specific number of random models, to letting the functions accept the number of models to create as a parameter. In this way, there is even more flexibility available to future testers and programmers while using the generators and factories. More current working changes and improvements, as well as other areas to look into will be discussed in Chapter 5.

4.3 Testing the Data

Once the factories were created, it became necessary of course, to test them. This section will serve as a brief discussion on how some of the testing was done, though it is certainly not exhaustive.

For the random generation function, testing was fairly easy. In those cases, either a number was passed in to determine the number of a models to be created, or a number was hardcoded into the generator, such as:

NUM_OF_PROFILES = 5

In either case, all that needed done was to compare the number of models in existence, to the number of models that should have been created. This was done by importing a test functionality:

from rest_frame.test import APITestCase

```
def test_generate_random_accounts(self):
    UserGenerator().generate_random_accounts(5)
    for ac in Account.objects.all():
        print(ac.account_status)

    self.assertEqual(Account.objects.all().count(), 5)
```

Figure 18: A simple test for testing the random generation of accounts

The APITestCase, then inherited by the testing class itself, allowed for an easy way to run and verify the success of tests, i.e., Unit Testing. Figure 18 shows an example of this in the testing of generating random accounts, where the number to be created is passed in. Unfortunately, because the data generated is itself random, it's impossible to test that the factory is generating the right sort of data in a unit test.

For the data that was generated from specific lists, the tests were both more complicated and more thorough. In most cases, it was easiest to iterate through the lists themselves and ascertain that the values in the list were indeed equal to the values in the objects. Figures 19 and 20 show code used for testing that agencies had been properly created. Notice that recursion is needed, since agencies can have child agencies, such that *Agency_1* might be the parent of *Agency_2* and everyone in *Agency_2* is in *Agency_1* but not everyone in *Agency_1* is in *Agency_2*. Thus, by recursively iterating through the list, we can ascertain that the agencies all have their assigned children, with the code in Figure 19 being called by the code in Figure 20 and then recursively calling itself until there are no more children.

```
def agency_models_testing(self, ag):
    # get db model for agency defined in list
    agency = Agency.objects.get(name=ag.get('name'))

    # get children of model defined in the list
    children = ag.get('children', [])

    # for each child of model def in list
    # test child exists in actual model in db
    for child in children:
        # print(child.get('name'))
        self.assertTrue(agency.children.filter(
            name=child.get('name')).exists())

        if 'children' in child:
            self.agency_models_testing(child)
```

Figure 19: The function used to help test agencies and called recursively

```
def test_create_model_agency(self):
    Generator().generate_agencies()
    self.assertEqual(Agency.objects.count(), 17)
    icac = Agency.objects.get(name="ICAC Data System")
    self.assertEqual(icac.is_visible, True)

    # dict of agencies
    test_agencies = [
        {"name": "ICAC Data System", "children": [
            {"name": "OJJDP ICAC Task Force Program"},
            {"name": "Fictional US BLE", "children": [
                {"name": "US West BLE", "children": [
                    {"name": "US Southwest BLE"},
                    {"name": "US Northwest BLE"}
                ]},
                {"name": "US East BLE", "children": [
                    {"name": "US Southeast BLE"},
                    {"name": "US Northeast BLE"}
                ]}
            ]},
            {"name": "Fictional Intercontinental Police", "children": [
                {"name": "Europe Police"},
                {"name": "Asia Police"},
                {"name": "South America Police"},
                {"name": "North America Police", "is_enabled": False},
            ]}
        ]}
    ]
    # iterate through dict
    for ag in test_agencies:
        self.agency_models_testing(ag)
```

Figure 20: The list of agencies and the start of the recursive call that tests them

Chapter 5: Conclusion

5.1 Solution's Success and Use

Using Factory Boy to synthesize fake data and creating *Generators* to then encapsulate the factories and create models in the various sections of IDS was very successful and moderately straightforward to implement. Within a few months, most of the necessary models for creating a test database that could be used for both testing and training were easily generatable without significant processing power. It was decided to create a simple command that a user could run that would create the entire fake database and run IDS. In this way, users could run the test server and database locally and work in the sandbox it provided, without any fear of accessing sensitive data or destroying necessary information.

Once it is fully implemented, the test system will allow for unauthorized developers to easily test and understand the system, demonstrations to be shown freely, and trainees to step through and work with fully synthesized and innocuous data. Further, if the models underlying the factories were ever changed, then most of the factories would change with them, and the *Generators* would still likely work, at least without significant edits. Thus, while there is still some work to be done in finishing the test database, the methodology of this project certainly led to a successful solution of how an LE database might be synthesized to allow for access by unauthorized users.

5.2 Future Work

While using Factory Boy for this project was successful, there is still room for future improvement and work on this topic. Much research exists on the study of creating feasible fake data for specific or simple types, but very little is actually geared towards faking specific data models or entire database tables. If a way could be found to automate the creation of complex data types and entire tables, it would certainly be useful in the field and world in general, given how omnipresent databases are.

One specific, and straightforward, way to improve on the project might be to find a better way for modeling many-to-many functionality in a more random and simple way. The methods inherent to Factory Boy, while not necessarily complex, are laborious and far from succinct. To create a better process by either overriding DjangoModelGenerator or some other method, would certainly be a worthy task. If it could be achieved where a many-to- many relationship might look something like:

profiles = factory.ManyToMany(ProfileFactory, random.randint())

where ManyToMany would define the attribute to contain a list of profiles, and random.randint() is a random integer, defined by the Python library *random*, it would certainly improve the process. Many-to-many functionality would then be built into the actual factory and significantly simpler and cleaner.

Another area where there is much room for improvement is the flexibility of this project. The methodology laid out in this paper was very focused on a specific system, and while it might be generally followed for similar systems, could not be easily copied. In fact, every system would need its own factories written, its own Generators created, and its own tests done. A way

to automate the process of, at least, factory creation, and perhaps Generator creation, would be far better. Given the need for systems such as the one that this project has designed, this area of research should be much further explored.

A suggestion for how to achieve this, might be to look at and mimic, override, or create something similar to Django's ability to auto generate models from a legacy database[6]. The method, called *inspect.db* creates models by introspecting an existing database. It will look through a database and make models based on what it finds. While the method is not perfect and the models created will need to be cleaned, it certainly is faster than creating every single model from the database by hand. Perhaps a method like this could be leveraged to generate factory models. Automated code could crawl through Django ORMs and generate the corresponding factories. Once finished, the factories would likely need cleaned manually as with Django's *inspect.db* method, but this would still be far faster than creating each factory model by hand.

Clearly, there is still much that can be explored in the area of database synthesis, where the desire is to create fake but realistic data for complex databases and models. As discussed, there are multiple methods for anonymizing the data that have been explored across the field. However, the creation of fake data has received much less attention, due to it being more complex in nature. While the methodology in this project has been successful, it was very specialized to the database being modeled. In the future, more work on automating this process should be done, and would be valuable in almost every field.

[6] https://docs.djangoproject.com/en/4.0/howto/legacy-databases/

References

[1] Z. J. Rashid and M. F. Adak, Test Data Generation for Dynamic Unit Test in Java Language using Genetic Algorithm," *2021 6th International Conference on Computer Science and Engineering (UBMK)*, pp. 113-117, Oct 2021. doi:10.1109/UBMK52708.2021.9558953.

[2] N. Senavirathne and V. Torra, "On the Role of Data Anonymization in Machine Learning Privacy," *2020 IEEE 19th International Conference on Trust, Security and Privacy in Computing and Communications (TrustCom)*, pp. 664-675, Feb 2020. doi:10.1109/TrustCom50675.2020.00093.

[3] S. S. Crossfield, K. Zucker, P. Baxter, P. Wright, J. Fistein, A. F. Markham, M. Birkin, A. W. Glaser, and G. Hall, "A Data Flow Process for Confidential Data and its Application in a Health Research Project," *PLOS ONE*, vol. 17, Nov 2022. doi:10.1371/journal.pone.0262609.

[4] W. -Y. Chen, M. Yu, and C. Sun, "Architecture and Building the Medical Image Anonymization Service: Cloud, Big Data and Automation," *2021 International Conference on Electronic Communications, Internet of Things and Big Data (ICEIB)*, pp. 149-153, Dec 2021. doi:10.1109/ICEIB53692.2021.9686426.

[5] Imperva, "What is Data Masking?: Techniques & Best Practices: Imperva," *Learning Center*, Oct 2021. Accessed: Apr 2022.

[6] "Data Anonymization: Use Cases and 6 Common Techniques," *Satori*, Nov 2021. Accessed: Apr 2022.