# Select, Insert, Delete

Now that you have declared models it's time to query the data from the database. We will be using the model definitions from the Quickstart chapter.

## Inserting Records

Before we can query something we will have to insert some data. All your models should have a constructor, so make sure to add one if you forgot. Constructors are only used by you, not by SQLAlchemy internally so it's entirely up to you how you define them.

Inserting data into the database is a three step process:

1. Create the Python object
2. Add it to the session
3. Commit the session

The session here is not the Flask session, but the Flask-SQLAlchemy one. It is essentially a beefed up version of a database transaction. This is how it works:

```
>>> from yourapp import User
>>> me = User('admin', 'admin@example.com')
>>> db.session.add(me)
>>> db.session.commit()
```

Alright, that was not hard. What happens at what point? Before you add the object to the session, SQLAlchemy basically does not plan on adding it to the transaction. That is good because you can still discard the changes. For example think about creating the post at a page but you only want to pass the post to the template for preview rendering instead of storing it in the database.

The **`add()`** function call then adds the object. It will issue an *INSERT* statement for the database but because the transaction is still not committed you won't get an ID back immediately. If you do the commit, your user will have an ID:

```
>>> me.id
1
```

## Deleting Records

Deleting records is very similar, instead of **`add()`** use **`delete()`**:

```
>>> db.session.delete(me)
>>> db.session.commit()
```

# Querying Records

So how do we get data back out of our database? For this purpose Flask-SQLAlchemy provides a **query** attribute on your **Model** class. When you access it you will get back a new query object over all records. You can then use methods like **filter()** to filter the records before you fire the select with **all()** or **first()**. If you want to go by primary key you can also use **get()**.

The following queries assume following entries in the database:

| *id* | *username* | *email* |
|---|---|---|
| 1 | admin | admin@example.com |
| 2 | peter | peter@example.org |
| 3 | guest | guest@example.com |

Retrieve a user by username:

```
>>> peter = User.query.filter_by(username='peter').first()
>>> peter.id
2
>>> peter.email
u'peter@example.org'
```

Same as above but for a non existing username gives *None*:

```
>>> missing = User.query.filter_by(username='missing').first()
>>> missing is None
True
```

Selecting a bunch of users by a more complex expression:

```
>>> User.query.filter(User.email.endswith('@example.com')).all()
[<User u'admin'>, <User u'guest'>]
```

Ordering users by something:

```
>>> User.query.order_by(User.username).all()
[<User u'admin'>, <User u'guest'>, <User u'peter'>]
```

Limiting users:

```
>>> User.query.limit(1).all()
[<User u'admin'>]
```

v: 2.x

Getting user by primary key:

```
>>> User.query.get(1)
<User u'admin'>
```

# Queries in Views

If you write a Flask view function it's often very handy to return a 404 error for missing entries. Because this is a very common idiom, Flask-SQLAlchemy provides a helper for this exact purpose. Instead of **get()** one can use **get_or_404()** and instead of **first()** **first_or_404()**. This will raise 404 errors instead of returning *None*:

```
@app.route('/user/<username>')
def show_user(username):
    user = User.query.filter_by(username=username).first_or_404()
    return render_template('show_user.html', user=user)
```

Also, if you want to add a description with abort(), you can use it as argument as well.

```
>>> User.query.filter_by(username=username).first_or_404(description='There is
```