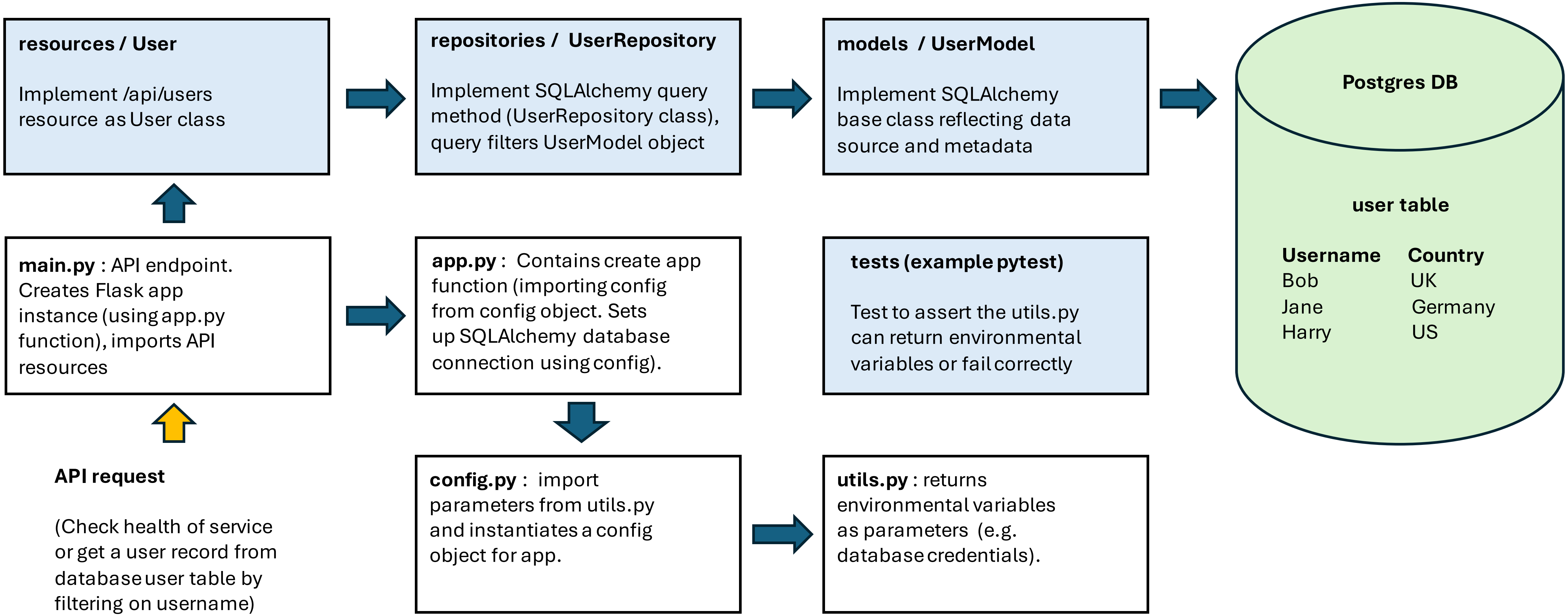
Basic example of accessing data using a RESTful API (SQLAlchemy and Flask)



resources / User

Implement /api/users resource as User class

repositories / UserRepository

Implement SQLAlchemy query method (UserRepository class), query filters UserModel object

models / UserModel

Implement SQLAlchemy base class reflecting data source and metadata

Postgres DB

user table

Username	**Country**
Bob	UK
Jane	Germany
Harry	US

main.py : API endpoint. Creates Flask app instance (using app.py function), imports API resources

app.py : Contains create app function (importing config from config object. Sets up SQLAlchemy database connection using config).

tests (example pytest)

Test to assert the utils.py can return environmental variables or fail correctly

API request

(Check health of service or get a user record from database user table by filtering on username)

config.py : import parameters from utils.py and instantiates a config object for app.

utils.py : returns environmental variables as parameters (e.g. database credentials).

GET *localhost:5000/healthcheck* request receives response body of {"status": "healthy"}
GET *localhost:5000/api/users/Bob* request receives response body of {"username": "Bob", "country": "UK"}

Implementing a repository design pattern for a RESTful API

A repository design pattern can be used as an intermediary layer between business logic and data storage for data access using an API. This layer separates concerns within a data access application by implementing separate layers for resources, repositories and models.

Resource layer

This layer will handle requests and responses (from a Flask app). It will interact with the repository layer (a query class) to perform database operations based on these requests:

`user = UserRepository.get(username)` from the username attribute of the /api/users API resource call the get method of the respository object created from the UserRepository object

Repository layer

This layer uses the SQLAlchemy library to query postgres data reflected in a SQLAlchemy Model class (UserModel):

`user = UserModel.query.filter_by(username=username).first_or_404()` populate a dictionary from the response to the SQLAlchemy query method which will form the API response

Model layer

This layer creates a Model class to reflect the metadata of a Postgres database schema:

```
username = db.Column(
    db.String, primary_key=True,
    unique=True, nullable=False)
country = db.Column(
    db.String, nullable=False)
```

Through this separation of concerns the application is easier to maintain and test. This application can be easily deployed as a microservice using containers and hosted on a cloud based container orchestration platform (such as Kubernetes or ECS). In AWS boto3 client logging can be streamed to CloudWatch for monitoring and alerting. The liveness of the service can be also polled using the /healthcheck API resource.