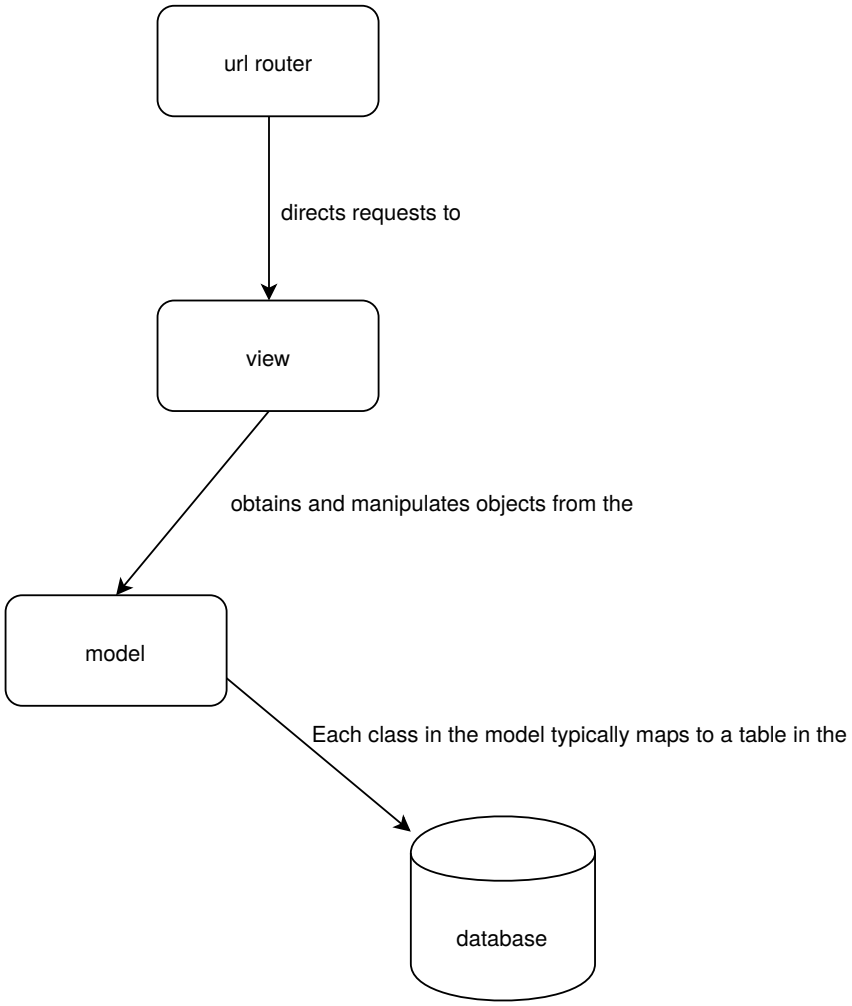
CITS5501 Software Testing and Quality Assurance
CITS5501
Web framework architecture

Web frameworks all have to solve similar sorts of problems, and typically encourage developers to structure their web applications into similar sorts of components.

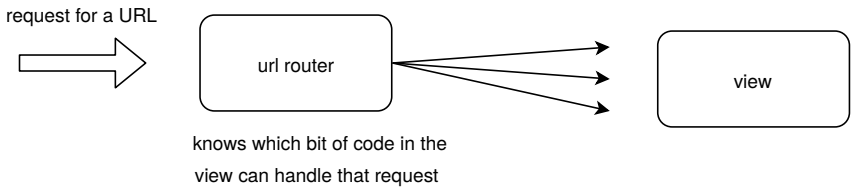
A typical architecture for a web application might look like the following (some details left off):



If you have no familiarity with how web servers or client–server architectures work, it may be useful at this point to refer to some of Mozilla's tutorials for beginners in web development: "What is a web server?", "How the Web works", and "Server-side website

programming".

URL router

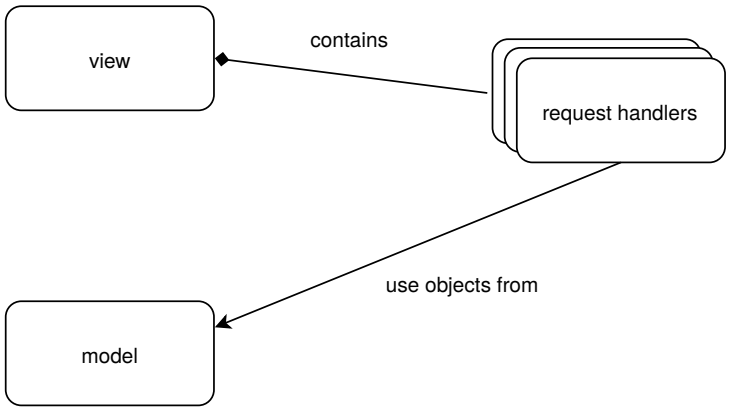


When a browser makes a *request* for a URL from the system, the URL router works out where to direct that request. A URL *path* (the portion of the URL after the hostname) to view the details of a staff member in an HR system might be something like `/personnel/view/some_staff_id/`; the URL router (which in Django is represented by the `urls.py` file) says which bit of code can handle this request.

If the request is for a URL the system doesn't know how to handle, there'll usually be a "catch-all" rule in the router that says to display an error page (e.g. a 404 "not found" page).

In broad outline, the router takes *in* requests for URLs, and says which bit of code (typically from the *view*) should handle that request.

View



The *view* component consists of individual *request handlers*. Each one knows how to handle a particular sort of request (e.g. a request for a page to view an employee record, add an employee, delete an employee, etc.), and has requests directed to it by the URL router.

A request handler is a function (or method, depending on the language) with a signature like this:

```
Response my_handler(Request req, /* ... other parameters ...*/ );
```

It takes *in* an object representing the browser request; works out what sort of HTML (or other data) it needs to return; and returns that data (sometimes wrapped up in a Response class).

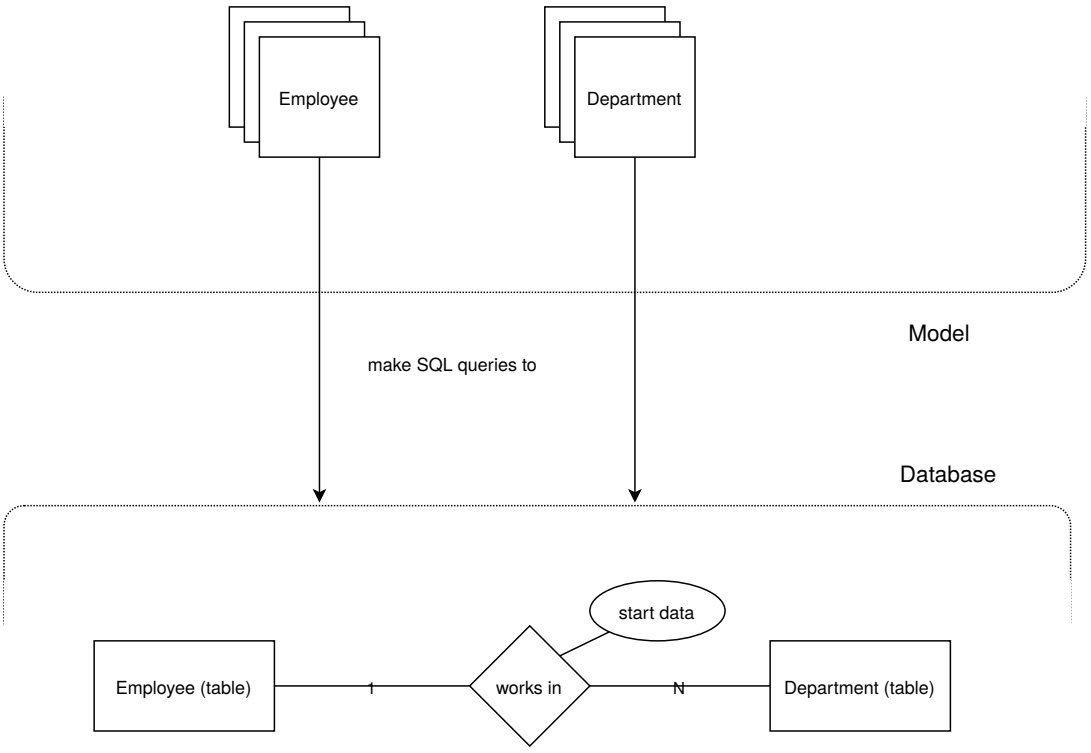
A Request object might be represented something like this:

```
class Request {
  String scheme; // were we asked for "http:", or "ftp:", or what?
  String path;   // the "path" portion of the requested URL

  // ... numerous other properties of the request
}
```

Most request handlers will query or alter data in the application *database*. However, they don't do it directly (by using SQL statements); normally, they manipulate objects from the *model*, which *represent* the data in the database. (The model objects act as a wrapper or "facade" around the database.)

Model



The "model" part of the application consists of a number of classes, with each class typically "mapping" to a table in the database. (Or more precisely, to an *entity* in the entity-relationship model the database instantiates.) For instance, in an HR web application, you might have classes such as `Employee`, `Department`, and so on.

For any sort of query or update you can imagine running on the database (e.g. "Get me a list of all employees", "Get me a list of all departments", "Get me a list of employees in departments X and Y", "Add employee A to department X") there will typically be a way of performing this using the model classes.

Testing objects from the "model" component

In order to test objects from the model component of the app, you'll need to create them, call methods on them, and see if those methods perform as expected. Since the "view" component of the app *also* needs to create objects from the model component and call methods on them, it follows that if you want to find out how to do this, then looking in the code for the "view" component is a good place to start. A quick Google search for "testing django models" should also provide you with some hints.