Backend Servers

Problem to solve

- Most apps don't live purely on your phone
  - Displaying/accessing content from other users
  - Backing up/syncing data across multiple devices
  - Accessing real-time or frequently updated data
- Typically we'll need some service that app users can access from anywhere

Server Requirements

- Support many simultaneous clients
- Encode/decode requests/responses
- Handle different types of request
- Store data persistently
- Handle authentication/authorization
- User session management

Common choice: HTTP

- Format used for requests/response is simple, and we can a variety of client types
- Straightforward way to differentiate the various request types: the URL path determines the type of request
- Flexibility in precise formatting: should response be JSON or HTML snippet or XML or some binary format?

Simultaneous clients

- The server will need to support many simultaneous connected clients, including concurrent requests from the same client
- There's many ways to handle this:
  - Thread per request?
  - Requests dispatched to a fixed sized thread pool?
  - Thread multi-plexing (`select` and similar functions to wait for any of a number of sockets)
  - Some combination?
  - Coroutines?
  - Multiple processes?

“Reactor” architecture

- NGINX is probably the first popular server platform to use this architecture.
- Most of the time our server is waiting for stuff (IO, DB query results) and likely performs very little CPU work
- Creating a bunch of threads which are mostly just waiting is inefficient
- The reactor pattern has a single thread listen for “events”, schedule a handler to deal with that, and then go back to listening for events

Reactor based HTTP file server:

```
while true:
    wait for an event
    if it's a new connection:
        register it so we events when there's bytes to read
    if it's bytes available on an existing connection
        read available bytes, assume it's an HTTP request
        start a "noblocking read" to read the contents of the requested file
        (we'll be notified when it's done)
    if it's a read completion:
        start copy the bytes we read from the file to the socket
    if it's a "write to socket finished" event
        close the socket
```

Reactor, key idea

- The thread running the event loop must never do anything that takes too long, otherwise events will come faster than we can service them
- If there is CPU heavy work to be done, we can have the event processing thread schedule it to run elsewhere
- Our library probably needs to let app developers write callback functions

  to be run when certain events occur. We could make it THEIR responsibility to never do anything slow on the main thread, or we could always run application code on other threads.
- If we choose option 1, usually the lib provides “nonblocking” functions for IO, DB access, etc
- You can imagine this leading to “callback” hell

Callback Hell

```
fs.readFile('/etc/passwd', (err, data)  > {
  if (err) throw err;
  slowMethod(data, (err, result)  > {
    if(err) throw err;
    use(result); //maybe this gets nested a few more times
  }
});
```

- Imagine if you need to do several slow operations in sequence. You'd get LOTS of nested callbacks
- try/catch won't work across callbacks, so you have lots of ugly explicit error handling!

Improved Reactor API

- Language support helps us make the developer API much nicer
- Basically async/await or “green threads” or coroutines all let app devs write code that looks blocking/synchronous, but which will actually yield whenever they call a function that would wait for IO
- For example, an app dev might write `row = someDatabaseQuery(x,y,z); return row[username]`
- The DB query call will actually set up the DB request, and then switch to some other task until the result is ready, then resume the task once it has the result
- The framework can have as many OS threads as it wants running and can schedule these tasks on them as it sees fit

Understanding Requests

- If we're using HTTP, request info could come in a few different places:
  - The URL/path part of the request (ex: /api/users/phoneNumber)
  - The query string part of the URL
  - The body of the request, common for POST requests. Could be JSON or “form encoded” `key1=value1&key2=value2`
  - Headers (possible, but I'm not aware of it actually being used)

Routing

- Most APIs based on HTTP encode most of the info about the request in the URL
- In general, a different “handler” function is called depending on the requested path, and method (GET, POST, DELETE, PUT, etc)
- If my API lets you retreive user email's by making a request to `/api/users/{some user id}/email`, then we need to “route” that paths that have some “wildcard” in them (the user ID) and to pass the user ID to the handler function
- You could use `url.split('/')` and some custom logic to handle this, but most frameworks provide library support for routing.

Response Encoding

- JSON is a convenient representation for sending data to the client (text based, human readable, etc)
- It's NOT a convenient representation for programmers (it's a string)
- Working with JS style objects is also not ideal (JS objects are basically HashTable, and we've talked about the downsides of that representation)
- Some frameworks will automatically convert normal objects into JSON when needed, so our function can return a `User` or a `CalendarEvent` and the framework will handle encoding it for us when needed

Data Persistence

- For any complex app with multiple users, we'll need the consistency guarantees of a database
- So any API server framework/library will need to provide support for at least one of them
- Maybe this is someting like a prepared statements API
- Hopefully it's something more like LINQ
- This will require some way of automatically converting between class definitions and DB schemas, and between DB rows and objects

Document based DBs

- Non relational DBs like Mongo and Firebase are essentially a hash table of JS objects which handle concurrency for you
- If you use one of these with a language like JS, you don't need a ton of library support, but you miss out on a lot of nice features or RDBMSs

Authentication + Authorization

- Your API server probably restricts what data is available/modifiable based on who's requesting it. A user can make posts to their profile, and admin can delete posts from anyone's profile, etc
- To support this, your server needs to support authentication: the user “logging in” and authorization: checking to make sure a user's request is allowed
- Since we're using HTTP, each request from the user is essentially starting a new conversation with the server, so how do we remember that they're logged in?

Cookies/Tokens

- Cookies are data stored persistently by the browser
- With each request, the client sends the values of its cookies via in the request headers: `Cookie: key1=val1; key2=val2`
- For authentication, typically the server gives the client some sort of session identifier which they include with each subsequent request as a proof of identity
- Some APIs use a different header field isntead of cookies for APIs (as opposed to browsers), for example the API for Canvas: `curl -H "Authorization: Bearer <access-token>" "https://canvas.instructure.com/api/v1/courses"`
- Your library/framework likely has an automatic way for manging this

External Authentication

- You might not want to handle authentication of user yourself, so you might use a scheme like OAuth
- You specify which “Identity Providers” you trust (Google, Facebook, Apple, Microsoft, etc)
- When a user logs in, you send them to the provider to log in, and if they're successful, the provider sends you back info about the user, if the login was successful
- Again, you probably don't want to deal with all this in hand-rolled code!

Session Management

- Once a user has authenticated, we want to keep them logged in for a while, but probably not forever
- We may also want to periodically change their secret “token” for security purposes
- Maybe we want to track data related to a user's activity on their phone differently from their computer
- These all fall under “user session management” which an app server framework should handle

Authorization

- Once the user is authenticated, each request needs to be checked to see if the user has access to the specified resource
- This might be through an API like `if(user.isAdmin){ ...}` or through an annotation like `@RequireRole(Roles.Admin)`, etc

More stuff: HTML Templates

- If your server needs to support browsers, it probably also needs to send HTML back for some requests, and JSON for others
- Often, we have a template of the HTML page and need to fill in a few blanks based on values
- There are tons of “templating languages” for writing an HTML skeleton and filling in blanks with variables

Some options

- There are a TON of options for libraries/frameworks for writing application servers
- Java: Springboot is probably the most popular, it also supports kotlin
- Kotlin: Ktor, from JetBrains, we'll look at this one
- NodeJS: ExpressJS, probably many others
- Go: The standard library does a lot of stuff
- Python: Flask, Django
- Rust: Rocket(?), Tokio is a low level event loop library
- D: Vibe.d
- Ruby: Rails
- C#: .NET ASP

How to choose?

- Probably pick your language and choose a library based on that
- Maybe you need a JVM langauge, or maybe you need native code
- Maybe your devs know python or JS already, maybe you want them to learn rust
- If you want to use a nosql DB like Firebase, a dynamic language is probably more convenient than a statically typed one
- Maybe you know that D is the best language

Examples

- We'll see a basic ktor project