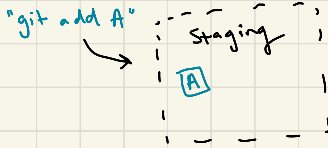
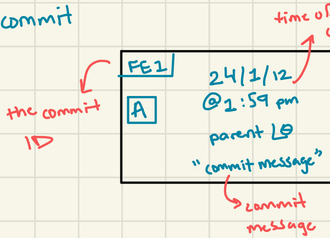
Git

Git subcommands

- $git [subcommand] [args...] • format
- $ git init • starts a new repo (for starting a project from scratch)
- $ echo "Hi" > A • prints the word "hi" onto a new (spontaneously created) file called "A"
  (as opposed to printing "hi" directly onto your console, which it would if we didn't specify a file)
- $ git add • adds files or directories to staging



- $ git commit • creates a snapshot (a "commit") of what is currently in staging

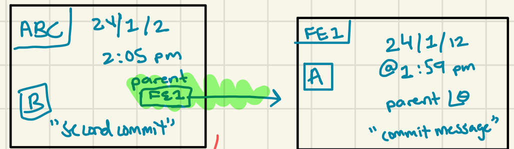


→ also stored in the commit:
- the author
- other details

- when a commit happens, the staging area also gets cleared.

Continuing the example

$ echo "B" > B
$ git add B
$ git commit -m "second commit"



→ all commits are connected through a graph/hashmap?

- $ git branch
  - what are branches? → "a reference to a commit ID"
  - what is the "head" (HEAD) of a git repo? → refers to the current branch you are "working on"
  - How do branches update? → when you make a new commit, the branch that HEAD refers to is updated to reference the new commit ID.

Class 1/17 : Branching in Git

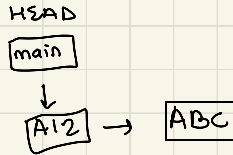
Why do we have to have a "staging" step?

→ So we have more control & can control exactly which files go into the commit.

What is HEAD?

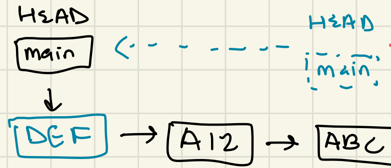
→ Refers to a branch ... indicates the one we're currently working on.

Example?



• here ABC was the 1st commit, followed by A12

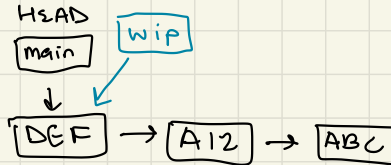
→ When we add a new commit called DEF:



the HEAD moves to reflect the new commit

$ git branch wip

→ creates a new branch named "wip":



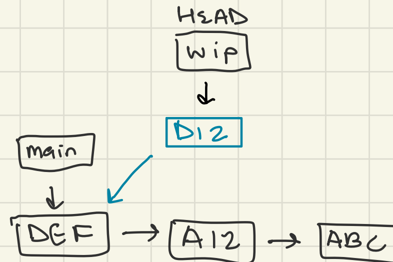
→ Big idea: we can have multiple branches. here, main and wip are 2 branches both referring to DEF

→ When we make a new branch the head does not move automatically

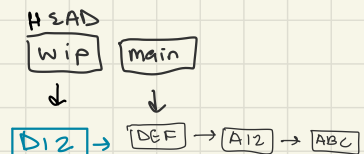
$ git switch wip

→ moves the HEAD to the branch "wip"

→ add new commit D12:



(or you could draw it like this :)



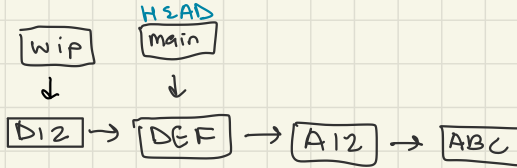
$ git checkout main

→ similar to switch except it would allow you to enter a detached head state (which switch won't)

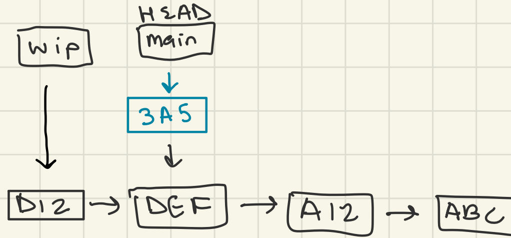
→ moves the HEAD to the branch "main":

$ git checkout A12

→ causes a "detached head state" because HEAD isn't referring to a branch (an error basically)



→ make a new commit with ID 3A5:



→ "The parent of 3A5 is DEF"

→ Then, we move the HEAD branch to the newest commit

→ In order: first add the commit in front of its parent. then, move the HEAD branch to follow / "track" the newest commit

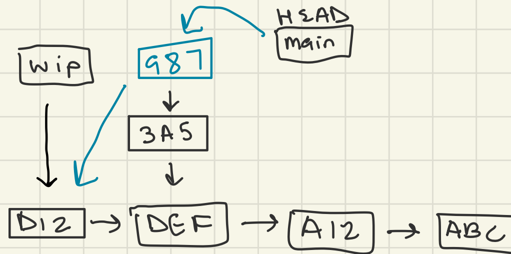
What is "merging"?

$ git merge wip

→ bringing branches together?

→ merges the HEAD branch (currently main) with the indicated branch (wip)

→ also, add new commit 987



-Strategies for lifelong learning in the software industry -

What are some key strategies?

→ Start with Office Tutorials and Getting Started Guides
  - written by creators of projects; helps you understand their focuses
  - written for a very general audience, typically professionals

→ Don't just glass over terminology you do not know; ChatGPT or Google it!

Typescript for the COMP301 Java Developer

→ The goal of this course is to teach the foundations of software engineering in the context of full-stack development

What is full-stack development?

→ involves both the frontend that a user interacts with (like a web/mobile application) as well as the backend that runs on one or more servers and serves data to applications across devices and many users.

What is JavaScript?

→ the leading prog. lang. for web applications.

What is Typescript?

→ a "superset" of JavaScript that adds static typing with optional type annotations to JavaScript

→ transpiles to JavaScript

→ used (in this class) to build the frontend of our web applications.

What does "static" mean in this context?

→ code written **at development time** rather than **at runtime**

→ kind of "code at rest"

- when it is in the editor or being analyzed by a compiler, rather than code that is actually running on a machine

What is static type specification?

→ allows the TypeScript IDE and compiler to verify that your code's expressions and statements are **type safe**.

→ decreases the risk of writing code that breaks after release (like when users are using it) rather than before.

→ serves as a form of built-in documentation for other ppl on your team to know how to use each other's code more reliably & confidently.

How does Typescript compare to Java?

→ similar in that it is also a high-level, OOP language ... but otherwise quite different

→ Syntax is more succinct & less verbose than Java.

– TypeScript Syntax –

What are primitive & reference data types?

→ primitive: a prog. lang's basic data types, from which all other data types are constructed.

- often denoted all-lowercase name
- RECALL: COMP 301 (Java's primitive types include int, double, & boolean)

→ reference: all the other types, often defined by interfaces, classes, & enumerations.

What are the primitive data types in TypeScript?

→ number: represents a number that can store fractions (decimal places)

→ boolean: T/F (same as Java)

→ string: represents a sequence of characters.

→ Unlike in Java, there is no type distinction between integers (int), floats, and doubles; number encompasses them all.

How do you declare a new variable in Typescript?

→ To learn syntax rules, we will compare Typescript (code written in blue) to Java (code written in orange)

Java

```
int newNum = 88;
```

Typescript

```
let newNum : number = 88;
```

- let → a keyword
- newNum → variable name
- : number → variable data type (this is the "type annotation")
- 88 → variable's value

→ The general format is let [var name] : [type] = [value] ;

How do you declare constants in Typescript?

→ RECALL: constants ≈ value can't be changed later

```
final int newNum = 88;
```

```
const newNum : number = 88;
```

const ↳ equivalent to "final" in Java

→ format: const [name] : [type] = value;

What do arrays look like in TypeScript?

→ TypeScript arrays are more similar to the Java List class than to Java arrays in terms of functionality (such as length not needing to be pre-set)

Java

```
List<String> names = new ArrayList<>();
names.add ("Avi");            —— add values
names.add ("Ella");
names.add ("Ameya");
names.set ("Rob", 2);         —— replace values
names.remove ("Rob");         —— remove values by value
names.remove (1);             —— remove values by index
String avi = names.get(0);    —— access values
```

TypeScript

```
let names : string [] = ["Avi", "Ella"]
```

- string → type annotation
- [] → declare arrays using [ ]

```
names.push ("Ameya");                    —— add values
names [2] = "Rob";                        —— replace values
names.splice (names.indexOf ("Rob"), 1);
names.splice (1, 1);
let avi : string = names [0];             —— access values
names.pop();
```

- remove values using splice (i, n), which removes n number of values starting at index i
- pop() → removes last item of an array

What do if-statements look like in TS?

→ same exact as Java!

```
if (condition A || condition B) {
      "    }
else {   "    }
```

What do while loops look like?

→ Same as Java:

```
while (conditionA) { ... }
```

What do for-loops look like?

→ similar to Java, with minor syntactical differences:

Java

```
for(int i=0; i<10; i++) {
    //
}
```

TypeScript

```
for (let i=0; i<10; i++) {
    //
}
```

→ for-each loops (iteration):

Java

```
for (String name : names) {
    //
}
```

TypeScript

```
for (let name of names) {
    //
}
```

- Functions in TypeScript -

What are functions?

→ The most fundamental abstraction technique used in software engineering

→ "self contained" modules of code that accomplish a specific task, usually by taking in data, processing it, and returning a result.

RECALL: How do functions look in Java?

→ Since Java is an OOP, function's can't exist on their own and instead exist as **methods** that are member's of some object class.
(even with anon classes & lambda expressions)

What is the difference between a function and a method?

→ **methods** are called on an object (eg object.method()), while **functions** are generally called standalone (eg function())

How do we write "functions" (methods) in Java?

```
String greet (String name) {
    return "Welcome, " + name + "!";
}
```

- String: the return type of the function
- greet: the name of the function/method
- (String name): the parameter input

How do we write functions in TypeScript?

```
function greet (name: string): string {
    return "Welcome, " + name + "!";
}
```

- function: keyword
- greet: name of function
- name: string: parameter input
- : string: return type specified @end (this is the type annotation)

→ general format: function + [name of function] + (param var's name: param data type): [returntype] { ... }

→ Unlike in Java, we do not need to specify/label a function as void if it doesn't return anything. We can, but it is optional;

```
function doSomething() { ... }
```

OR

```
function doSomething(): void { ... }
```

What are arrow functions?

→ A more compact and concise method of defining traditional functions

→ basically **saving a function to a variable** with a name that we can then use to call it.

Example?

using the "let" keyword to assign the function as a variable

```
let greet = (name: string): string => {
    return "Welcome, " + name + "!";
}
```

the variable itself will be a string (b/c that's the return type), but "=>" is used to connect it to a function body

→ Arrow functions can then be called normally:

```
let avi: string = greet("avi");
```

("Welcome, avi!" is now the value of the variable avi)

Why are arrow functions significant?

→ **They open the door to a whole new world of programming called functional programming, since we can now pass functions around as values!**

What are some key differences between arrow & traditional functions?

1. Arrow functions do not have their own this. bindings, & thus shouldn't be used as class members/methods (b/c you can't call "this.greet")
2. Arrow functions cannot be used as constructors (can't call them with "new")
   - would throw a TypeError

– Class and Interface construction in TypeScript –

→ The core idea and motivation for classes & interfaces in TypeScript is the same as that of Java (basically what we learned in 301)... mostly just syntax differences.

→ Lets compare an example class construction in Java vs Typescript:

- green boxes highlight minor syntactical differences

How do we define class fields?

Java

```java
public class Student {
    public String name;
    public int year;
    private String address;
```

Typescript

```typescript
public class Student {
    public name: string;
    public year: number;
    private address: string;
```

let keyword not used when defining fields

→ SAME - access modifiers: public, private, & protected keywords mean the same thing in both langs.

defining the constructor?

Java

```java
public Student (String name, int year, String adr) {
    this.name = name;
    this.year = year;
    this.address = adr;
    this.welcome(); }
```

Typescript

```typescript
constructor(name: string, year: number, adr: string) {
    this.name = name;
    this.year = year;
    this.address = adr;
    this.welcome(); }
```

→ SAME - use of the "this" keyword

defining class methods?

"void" keyword optional

"function" keyword not used when defining functions as methods

```java
public void welcome(){
  System.out.println("Hello," + this.name);
  }
public static String yrToString(int year){
  if(year == 1){
    return "Freshman";
  else if (year == 2){
   return "sophomore";
   }
  return "Oops...";
  }
  }
```

```typescript
public welcome(){
  console.log("Hello," + this.name);
  }
public static yrToString(year: number): string{
  if(year == 1){
    return "Freshman";
  else if (year == 2){
   return "sophomore";
   }
  return "Oops...";
  }
  }
```

→ SAME — meaning of static keyword.

How do we instantiate a new object in TS?

→ Same as Java, except for syntax with variable declaration:

```
Student noah   } = new Student("Noah", 3, "Columbia St");
noah: Student  }
```

How do we create & implement an interface in TS?

```typescript
public interface Person{
  name: string
  }
```

```typescript
public class Student implements Person{...}
```

What is structural type checking?

→ A feature of TS where it views objects as being equivalent types if they share the same structure, regardless of whether they share the same type name.

→ This means that we can technically directly create a variable of an interface type (like Person), without needing to have or use a class that implements the interface!

```typescript
let person1: Person = {
  name: "Avi"
 };
```

creating object of type Person (an interface)

defining the properties required/outlined by the interface

• notice how no subclass was used or mentioned in this declaration •

Does Java have this feature?

→ No! Java, on the other hand, is a nominally type language, meaning it views objects as equivalent types iff. they share the same name (or have an inheritance relationship)

• RECALL how you can only create objects of interface type if an implementing class is specified:

```java
Person avi = new Student(...);
```

What is the idea behind structural type checking?

→ it relaxes the strictness of nominal typing (like Java's) by embracing the idea that if an object has all the same fields & methods needed as some other object type, then its probably OK to treat it as that other type.

-Extra TypeScript Features and Syntax-

How do you print statements in TypeScript?

→ using `console.log(...)`, which is the TS equivalent of `System.out.println(...)`.

How do you use enums in TS?

→ (RECALL 301) the same as Java, except that the enum options only have their first letter capitalized, rather than all caps.

What is a type alias?

→ a feature of TS that allows you to create another **label** by which you can refer to some object type.

→ Basically giving a known (primitive or reference) data type an alias name that can then be used interchangeably with the type name (when creating & working with the objects)

How do you create a type alias?

→ Using the `type` keyword:

```
type Rating = number;
let csxlRating : Rating = 10;
```

- `type Rating = number;` ← creating an alias called "Rating" for the number data type
- `let csxlRating : Rating = 10;` ← creating new obj of type "Rating" (but its really just a number)

Why are type aliases useful?

→ To make types more concise or more readable for your feature

→ basically just an aid for organization of code?

What is a ternary operator?

→ an operator (like &&, ||, etc.) that allows you to write a **conditional expression**.

- as opposed to the other operators, which are only statements, and you have to write subsequent code to produce an expression based on their result (using if/else statements, for ex)

→ basically, you input a T/F condition into the ternary operator. If it is true, the expression can evaluate to one value. and if its false, another.

How do we write ternary operators in TS?

→ using this syntax: `[condition] ? [expression if true] : [expr if false]`

→ Example:

```
let csxlOpenHour : number = isWeekday ? 10 : 12;
```

this is the ternary operator: `isWeekday ? 10 : 12;`

```
console.log(csxlOpeningHour);
```

- if the boolean "isWeekday" = true, the console will print **10**
- if "isWeekday" = false, the output will be **12**

```
console.log(isWeekday ? "Weekday" : "Weekend");
```

- since the ternary operator produces an expression, it can also be used on its own like this
- the "expression" can be any data type, not necessarily number

RECALL: What are generic types?

→ A powerful convention in Java (and also TypeScript!) that allows you to pass types as a **parameter** into objects. This creates objects that can support multiple data types.

How do we create generic types in TS?

→ basically the same as Java, just syntax differences.

→ see reading for comparison of generic class implementations in TS and Java.

Java

```java
LinkedList<String> myStList = new LinkedList<>();
LinkedList<Student> myRoster = new LinkedList<>();
```

TypeScript

```typescript
let myStList : LinkedList <string> = new LinkedList<>();
let myRoster : LinkedList <Student> = new LinkedList<>();
```

Introduction to Higher Order Functions

RECAP: What does a function look like in T.S.?

→ Traditional Function:

(annotations: parameter → `num: number`; return type → `: number`)

```
function doubleNumber (num: number) : number {
    return num * 2;   }
```

→ Arrow Function:

```
let doubleNumber = (num: number) : number => {
    return num * 2;   }
```

→ arrow function syntax uses the same syntactical structure that we use to define variables!

What are the implications of arrow functions?

→ Functions now join the list as a possible type of value that can be stored in a variable:

Variable	Value	Value's type
let course = 423;	423	number
let name = "Kris";	"Kris"	string
let yoda = new Jedi("yoda");	Jedi("yoda")	Jedi
let doubleNum = (num: number): number => {return num * 2;}	[ ]	"(num: number) => number"

What are function literals?

→ a reference value that declares an anonymous function

→ defines the 3 basic parts of a function — the [1] parameters (inputs), the [2] return type, and the [3] function body.

→ Syntax:

```
1 — (num: number) : number  => {      ← 2 (return type)
        return num * 2;   }            ← 3 (function body)
```

RECAP: What are type annotations?

→ type annotations are a way to explicitly specify the expected data type of a parameter, return val, or any other variable being declared in a program.

- In TypeScript, type annotations are optional (RECALL the code from EX00) because TS is usually able to infer the expected data type.
- However, they are useful to the compiler in checking types, and can help avoid errors dealing w/ data types

What is the type annotation for normal variables?

→ Followed by the variable name, in the format of " : type " :

with type annotation	without type annotation
let name : string = "Avi";	let name = "Avi";
let PID : number = 7305;	let PID = 7305;

→ Unlike Java, TS is able to read this and infer that name is meant to be a string

What is the type annotation for functions?

without type annotation

```
let doubleNumber = (num: number): number => {
    return num * 2; }
```

with type annotation

```
let doubleNumber : (num: number) => number = (num: number): number => {
    return num * 2; }
```

What are some use cases for passing functions around as values?

1. Passing Functions as parameters
2. Returning Functions from other Functions

– Passing Functions as Parameters –

The example:

→ Imagine we have several arrow functions that take in a number and multiply it in some way – similar to doubleNumber ex above ⤴:

```
let doubleNumber : (num: number) => number = (num:number): number => { ... }
let tripleNumber : (num: number) => number = (num:number): number => { ... }
let halveNumber : (num: number) => number = (num:number): number => { ... }
let squareNumber : (num: number) => number = (num:number): number => { ... }
```

How can we create a function to call doubleNumber on every number in an array?

→ lets create an arrow function called mapNumbers:

```
let mapNumbers : (nums: number[]) => number[] = (nums: number[]): number[] => {
    let newNums: number[] = [];
    for (let num1 of nums) {
        let nextNum = doubleNumber(num1);
        newNums.push(nextNum);
    }
    return newNums; }
```

(type annotation telling us that both the parameter arg and return type will be number arrays)

calling "doubleNumber" on each variable in the parameter-passed array, and adding the result to a new array.

What would make mapNumbers a more multipurpose function?

→ By making it able to not just double a number[], but square or triple or halve it too!

→ We could do this by just rewriting mapNumbers every time, but that would be inefficient

How can we do this efficiently?

→ by passing in the specific function (doubleNum, tripleNum, etc.) that we want performed on each number!

→ Since functions can be used as values, we can use them as function parameters too!

- Notice the difference: in Java, there is no way to pass a method as a parameter into another method.

Syntax to pass in a function as a parameter?

→ With the same format as any other variable being passed in; "<name> : <value type>"

```
let mapNumbers = (nums:number[], transformFunc: (num:number) => number ): number => {...}
```

a param. arg named "nums", of a number array value type

a parameter arg named "transform Func" of value type "(num:number) => number", meaning a function taking in a number & returning a number

→ Basically, a function can take in a **certain type of function** as its parameter

- this "type", as we can see, is defined by the param. & return types.

How can we make our code more concise?

→ Passing in certain types of functions can be wordy, but we can fix this by creating a **type alias** (recall: RDOO)!

```
type numberTransformer = (num: number) => number;
```

What would the mapNumbers implementation look like now?

```
let mapNumbers = (nums :number[], transformerFunc: numberTransformer ) :number[]=>{
    let newNums :number[] = [];
    for (let num1 of nums) {
        let nextNum = transformerFunction (num);
        newNums.push (nextNum);
    }
    return newNums;  }
```

numberTransformer ← type alias

transformerFunction (num) ← calling the passed in function!

-Returning Functions from Other Functions -

→ since functions can now be used as values, they can also be the return type of another function!

Example of when to use this?

→ Recall the functions we created and used as parameters in the previous example:

- doubleNumber
- tripleNumber
- squareNumber
- halveNumber

→ Instead of manually creating & implementing these functions that all effectively follow the same pattern — returning their input number by some factor x (2, 3, 1/2, or in the case of squareNumber, the input num itself) . . .

- What if we could create a function that generates diff functions based on the factor we want to multiply by?

→ RECALL the **factory design pattern** — creating diff versions of a class based on a certain parameter.

How would we write such a function?

```
let generateMultiplierFunction =
(factor : number) : ((num: number) => number) => {
```

(takes in the desired multiplying factor)

(RETURNS a function of this specific type, aka one that is compatible with mapNumbers)

```
    let ret = (num: number) : number => {
        return num * factor; }
    return ret;
}
```

(creating & returning a function)

→ Now, we can call mapNumbers even more efficiently:

- Triple all numbers in an array:

```
let numList : number[] = [0, 1, 5];
let tripledList : number[] = mapNumbers (numList, generateMultiplierFunctions (3));
```

- "tripled List" : [0, 3, 15]

- Conclusion -

What are "higher order functions"?

→ Functions that either:

- * take in other functions as parameters, or
- * return a function as their result

Why are they useful?

→ Enable us to program more in a functional programming style, by allowing us to abstract functionality into higher order functions.

Learn a Command Line Interface Ch1: The Sorcerer's Shell

What is "learncli $"?

- → What you see as the last line in your terminal
- → a bash command-line interface (cli) prompt, also known as a shell prompt.

How does the CLI work, on a broad level?

- → basically, you type a command into the shell prompt (there are MANY, we will learn about them), followed by whatever parameters/specifications the command dictates.
- → then, you press enter and the CLI reads your command, interprets it, and attempts to carry out your request!

What is the "ls" program?

- → A standard utility program found in the /bin directory
- → The sole purpose of the ls program is to list the files contained in directories.

How do you use the ls program?

- → By typing in the command, "ls", followed by (a space & then) the name of the directory (in the format /<directory name>)
  - the CLI will return a list of all files contained in that directory.

Example of using ls?

TERMINAL

```
learncli$  ls  /bin          → hit ENTER & this appears:

bash      dd      launchctl   pwd        tcsh
cat       df      link        realpath   test
chmod     echo    ls          rm         unlink

  [...]
```

What is the bin directory?

- → "bin" - short for "binary program files"
- → Stores files which your computer can evaluate as a program - like ls!

How do you learn what a command line program is useful for?

- → run the program in "help mode" by typing the program name, followed by the -help argument:

  ```
  learncli$  ls -help
  ```

- → this command prints out a bunch of text that contains info about the program's purpose, usage, and options.
- → -help usually prints out a lottt of info - so much so that the text might scroll off of your screen.

How do we see less text output at a time?

- → With the less program!
- → Type `ls -help | less` to only see a single screen of output at a time.
- → Keyboard Shortcuts in less:

Key	Motion
f	page down
b	page up

Key	Motion
j	scroll down (by line)
k	scroll up
q	Quit

What is a "pipe" in the Unix command line?

→ A way to connect programs together, that connects the output of one CLI program to the input of another — leading to a multiplicative effect on the no. of tasks you can carry out.

→ represented by the vertical bar character, | -- like when we did `ls --help | less`!

→ Pipes are part of an important Unix C.L. concept called **composing programs**.

What does `man` do?

→ A program to read the manuals of other programs. Running the command

```
learncli$ man ls
```

replaces the terminal's content with the manual for the `ls` program.

What is in the manual?

→ All of the info in a given program's `help` mode, and more!

→ Unlike the text that comes up when you do `-help`, manual pages have consisted, improved formatting already organized into pages.

- Scroll the manual using the same keys used to navigate `less` (prev page)

What is the `cat` program?

→ reads data from a file and gives/prints its content as output.

```
cat /usr/share/dict/words
```

What does the TAB key do?

→ When you are typing in a command and press TAB, the shell autocompletes whatever it thinks you are trying to type - or gives you several options if there's more than one possibility.

- Similar to the thing on iPhones where 3 boxes of word options come up while ur texting
- especially useful when typing in the file path of a file.

What do the up & down keys do?

→ flips back & forth between previously used commands in your history (so you don't have to retype them if you want to reuse them)

What does the `clear` program do?

→ typing in the `clear` command clears your terminal screen & resets the `learncli` prompt to the top of the terminal.

What is the `grep` program?

→ uses textual patterns to search for textual matches ... basically like ⌘F but way cooler!

→ the command follows the format

```
grep [OPTIONS] PATTERN [FILE...]
```

- PATTERN: the string of characters that you are searching for
- brackets indicate optional arguments that can be left blank
- ellipsis indicates you can list multiple files (separated by spaces), and grep will search all of them

→ `grep` prints out all lines of every listed file that contain a textual match.

Examples using grep?

- learncli$ grep motion /usr/dictionary
  - commotion
  - demotion
  - ....
  - prints all lines of the dictionary containing the string "motion"
- learncli$ grep ^motion /usr/dictionary
  - motion
  - motion's
  - motioned
  - ...
  - the ^ character anchors the pattern to the start of a line!
  - returns all lines beginning with "motion"
- learncli$ grep ^g..p$ /usr/dictionary
  - gasp
  - glop
  - goop
  - gulp
  - ...
  - the $ character anchors the pattern to the end of a line!
  - the . character matches "any character"... so basically used as a placeholder if you want to specify how long the search string will be.
  - returns all strings beginning with a g, ending with p, & with 2 characters in between.

What type of program is grep?

→ A command-line program that **filters data**.

→ Such programs tend to operate in one of two ways:

1. accept a list of files to process (like in the examples above)
   - this is a convenient but not all-that-significant feature
2. operate on data piped into them by other programs!
   - An essential & more powerful feature/usage of grep.

Examples of using grep with pipes?

- learncli$ ls /bin | grep ^g..p$
  - grep
  - gzip
  - Instead of providing the optional [FILE...] argument, we have grep search the output of "ls /bin" — which is the list of file names inside /bin
  - The command returned the only lines/file names matching the specified argument.

→ cat is an especially useful program to use in conjunction with grep, as we can search the contents of a file being read by cat

What does the command "history" do?

→ prints a list of the trail of commands you have recently run — aka your command log.

Ch. 2: Directories, Files, and Paths

What is a file system?
- → a way to organize all of your projects & other work in **files and directories**
- → The Finder application on a Mac is a GUI-based way to navigate your file system, where you can search, organize, rename, etc. files all just by pointing and clicking.

So why should you even try to navigate your file system via the CLI?
- → Although it takes more effort to learn, it provides you with a LOT more power.
- → Using a CLI, you can easily **automate** repetitive file system tasks, such as renaming 1000s of files from one naming convention to another
  - something that would likely take days or hours to do via the GUI

What is a directory?
- → the fundamental unit of organization in a file system
- → Every directory can contain files, as well as other directories in a hierarchical relationship
- → there is one **root directory** that has all other directories & files as its descendants
- → "directory" - synonymous with "folder" (like in GUI style view)

How do you access the list of contents in the root directory?
- → RECALL: the program to list the contents of a directory is `ls`. To list the contents of the root directory, use `learncli$ ls /`
- → The forward slash `/` is how you refer to the root directory!
  - `/bin` = The `bin` directory, located in the root directory
  - `/usr/share` = File path for the `share` directory, located in the `usr` directory, located in the root directory!

  And so on...

What is a path?
- → The textual "address" of a directory or file in the file system.
- → When wanting a program to operate on a file (like `grep`), you provide the file path as an argument!

What is an absolute path?
- → Paths which begin with a forward slash, referencing the root directory.

What is the basename?
- → The last name in a file path, which is the file/directory that the path is specifically referring to — the "target" of the path.

What is the "dirname" of a path?
- → Everything that comes before the basename, including the forward slash.
- → Represents the 'path' that is leading you to the target.

dirname	+	basename	=	absolute path
/usr/share/dict/		words		/usr/share/dict/words

What is a "working directory"?
- → When you need to work on many files in a single directory, typing all of their absolute paths all the time would become tiring.

→ Instead, we can tell the shell that that directory is our working directory, and then only need to write shorter, less redundant "relative paths" to files in it.

What does the pwd program do?

→ Prints the path of your current working directory!

→ The shell already maintains a current w.d. as part of its state (though we can easily change this). Currently, learncli211% prints out

/Users/avikumar/learncli211

How do we change our working directory?

→ With the cd command, followed by the filepath to the directory we want. c.d. = "change directory"

```
learncli$ cd /usr/share/dict
learncli$ pwd
/usr/share/dict
```

← changed the w.d. – confirmed with the pwd command's printout

What happens if you type ls without any proceeding arguments?

→ It prints the list of contents of the current working directory, by default.

```
learncli$ ls
american-english  words
```

What is an example of how a w.d. makes it easier to type commands?

→ If we want to use cat to print the contents of a file, we can now simply use the relative path to refer to the same file – since its absolute path has already been specified:

```
learncli$ cat /usr/share/dict/words | less
```

(RECALL ch.1)

VS

```
learncli$ cat words | less
```

↑ the 'relative path'

When should we use each type of path?

→ either kind of path – relative or absolute – can be used anywhere that a path is expected! You can freely substitute absolute with relative paths, and vice versa.

How does the learncli container work?

→ The learncli container's file system is separate from that of my PC, meaning that changes I make in my container's file system all revert back to their original state whenever I exit my learncli session (w/ exit command) – which is good for when I make accidental changes.

What about the learncli directory?

→ HOWEVER, the actual /learncli directory is different – it belongs to my computer's file system (you can literally open the learncli211 folder on Finder by going to avikumar → learncli211)

- this means that all files within it are accessible & modifiable by my PC

→ The /Users/avikumar/learncli directory is "mounted into" the learncli container.

Question	Notes
	learncli211 % ls
	LICENSE lab-00-aviomg ssh
	a.out learncli.ps1 workdir
	bin learncli.sh
	→ Typing open . into the learncli211% prompt opens the corresponding directory in the GUI (Finder application)
How do you create new directories using the terminal?	→ With the CLI mkdir program & command
	→ Makes a new directory inside of the current w.d.
	learncli$ cd workdir
	workdir% mkdir ch2 ← creates new directory called "ch2" inside of "workdir"
How do you copy files in the terminal?	→ With the cp program/command! We can make a copy of a "source" file & place it in a "target" file or directory.
	→ There are 2 ways to use the cp command:
	1. cp + [path of SOURCE file] + [path of TARGET file]
	• copies contents of one file into another
	2. cp + [path of SOURCE] ... [path of TARGET directory]
	• creates copies of source file(s) & adds them to target directory
	• the ellipsis means we can list multiple src files, separated by spaces
	learncli211$ ch2 % cp /usr/share/dict/words words
	• copies the content of ↗ into a file called ↗, in our ch2 directory.
What does the cp --recursive option do?	→ Copies entire directories & their contents.
What is --verbose ?	→ An argument that you can use when running programs that will cause them to print out the actions it performed when you ran it.
	• basically if you want to know exactly what the program is doing
	→ On Mac terminal, enter this argument as the flag "-v":
	mkdir -v practice-directory
	mkdir: created directory 'practice-directory'
What are "hidden dot files"?	→ Files and directories which begin with a period, ".", and are considered "hidden" files — they aren't displayed when listing a directory's content with ls.
	→ Typically used to store the settings, preferences, and metadata of tools & projects.
	cp words .words-copy
How do we ask ls to list hidden files?	→ With the -a or -A arguments
	→ -A lists all hidden files except "." and ".."
	learncli211$ ch% ls -a ⇝ . .. .words-copy a-sub-dir words

What is a "link"?

→ A third kind of file system entry (besides a file or a directory)

→ A link "points" to something else in a system

What is ".."?

→ A link that automatically exists inside every directory

→ .. is the parent directory link, & points to the parent directory of your current w.d.

What is "."?

→ Another link that automatically exists inside every directory and links to itself (aka "points" to the current working directory)

What is the point of the . and .. links?

→ They basically provide a shorthand to make typing commands more efficient (sort of like `this` in Java)

→ . is particularly useful when you want to specify the current directory as an argument to a program which is expecting some directory's path.

Usage examples?

→ You can move to your parent directory using `cd ..` instead of `cd [name of parent dir]`

→ To create a relative path to move "up" the file system hierarchy by more than one p.d. at a time:

```
learncli211$ a-sub-dir% cd ../..
learncli211$ workdir %
```

moved from a-sub-dir to ch2 to workdir

→ To copy a file from one directory to another with `cp` & retain the same file name:

```
learncli$ cp /usr/share/dict/american-english .
learncli$ ls
    american-english
```

How do you rename files in a CLI?

→ With the `mv` program/command!

→ `mv` + [path of SOURCE file] + [new desired name]

```
learncli $ mv .words-copy words-copy
           ls -A
a-sub-dir  american-english
words      words-copy
```

here, we moved the file named ".words-copy" to the name "words-copy"

How do you move files from one directory to another?

→ Also using the `mv` command, except instead of a "new desired name", the 2nd argument should be the directory where we want to move the file:

```
learncli $ mv words-copy a-sub-dir
           ls
a-sub-dir  american-english  words
           ls a-sub-dir
words-copy
```

notice how words-copy no longer shows up in the ls of the cwd ... but it does show up in the ls of the target, a-sub-dir!

What does the find program/command do?

→ It lists directories recursively – aka, it lists the name of all files and subdirectories in a given directory (which you specify as an argument), but below each listed subdirectory, it also lists the names of the files inside it!

→ As opposed to the ls command, which only lists the name of everything in a given directory (but not the content inside any subdirectories)

How do you use find?

find + [path of starting-point directory]

What is an example of using the find command?

→ Say you have the following content inside your ch2 directory:

words (file)	a-sub-dir (directory)	practicedir (dir)
words-copy (file)	↓ words (file)	book (file)

→ Using ls returns a list of directory contents:

```
learncli211$ ch2 % ls .

words   words-copy   a-sub-dir   practice dir
```

→ Using find returns a recursive list of directory contents:

```
learncli211$ ch2 % find .
.
./words
./a-sub-dir
./a-sub-dir/words
./practice dir
./practice dir/book
./words-copy
```

(notice the . referring to the cwd, "ch2")

(notice the . being used as a shorthand for "ch2" in the relative path names)

How do you delete files from the command line?

→ With the rm program/command!

→ rm will only delete files, **not** directories – unless you use the -d argument.

```
learncli211$ ch2%. rm words
```

How do you delete empty directories?

→ Using rmdir

```
learncli211$ ch2%. rm a-sub-dir
```

How do you delete non-empty directories?

→ Using rm but with the -r argument, which stands for "recursive", meaning it tells the rm program to traverse all subdirectories to delete files.

→ Be CAREFUL when running rm recursively – use the -i argument, which results in the terminal asking you to confirm, for each file, that you want it deleted (you just type "y" or "n" to confirm or deny).

Functional-Style Stream Processing

- → A powerful model for processing events & data
- → Applications: when you want something to constantly update, like the scores for a sports game on a website

What is stream processing?

- → A technique spanning many areas of CS from big data processing in the back-end, to event processing in the front-end and more.
- → Historically, functional-style prog concepts have played a big role
  - pure, 0-side effect functions & immutability enable parallelism/scaleability
  -

What is a stream?

- → Basically a collection of data

What is a functions type?

- → generally, any 2 values of the same type can be substituted for one another & the program's type checking will remain valid.
- → is made of its parameter list's types tuple & its return type

  ($parameterType_0$, ...) : returnType

What is a functional interface?

- → Assigns an <u>identifier</u> ("name") to a type of function.

```
interface Name {
    (parameter0: type0, ...): returnType;
}
```

- → For ex, this says "A predicate is any function with a single type of number that returns a boolean."

```
interface Predicate {
    (element: number): boolean;
}
```

Functional Interface Types with Generics

- → with diamond notation:

```
interface Transform<T, U> { (param0: T): U; }
```

- → specifying a generic type is "parametrizing a type" to avoid redundant declarations of types with the same "shape" for each type involved.
- → Generic types can be made concrete in subsequent declarations of types of variables, parameters, & return types. For example:

```
let stringtoInt: Transform<string, int> { (param0: string): int; }
```

→ Modern languages' ability to infer types improves the

Type Inference

→ Conversely, type inference works in the opposite direction as well with "contextual typing"

```
let f1: Predicate<number> = (x) => { /*...*/ };
```

↑ here, the parameter & return type are inferred by TypeScript

→ Contextual typing makes the DX of functional-style programming with higher-order functions incredibly pleasant & powerful...

```
interface Transform<T, U> { (p0: T): U; }
let map = <T, U>(collection: T[], f: Transform<T, U>):
    U[] => {
        let rv: U[] = [];
    for (let item of collection) {
        rv.push(f(item));
    }
    return rv;
    }
```

Aside: Anonymous Function short-hand

→
```
let f: Transform<number, string> = (x: number): string => { return x + "!"; }
```

SHORTEN ↓

```
let f: Transform<number, string> = (x) => { return x + "!" };
```

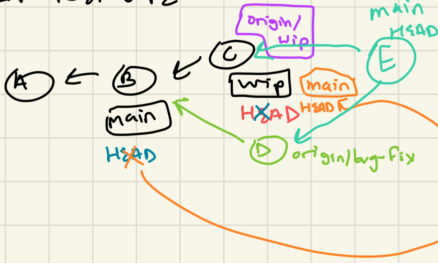
↓

```
let f: Transform<number, string> = (x) => x + "!";
```

↑ Since body is a one-liner (just return statement)... we don't need return keyword.

```
let f: Transform<number, string> = x => x + "!";
```

GIT REMOTE



git push origin wip
↳ creates a new branch in the GITHUB physical server

git fetch --all ← retrieves & downloads all of your remote branches (so bug-fix)

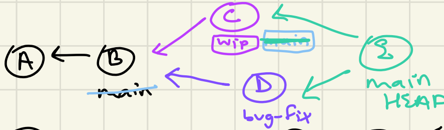
git switch main

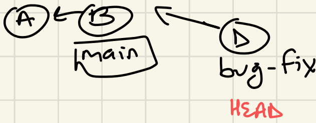
git merge wip

git push origin main
(a "Fast-Forward")

git merge origin/bug-fix

git push origin main





git push origin bug-fix

git switch main

git pull origin main

1) push bug-fix
2) push wip
3) merge

→ New Pull Request: bug-fix → main

- would have to approve this on github
-

Overview of Angular & Components

What is Angular?
→ A development platform built on TypeScript

What is a component?
→ Components are the fundamental building block for creating Angular applications.
→ define areas of responsibility in the UI, and allow you to reuse sets of UI functionality.
→ Make a new component in your application by running `ng generate component <componentname>` in the container terminal.

What happens when you generate a new component?
→ The 3 files which compose a **component** are created:

1) a component class, written in TS, that defines the **behavior** of the component.
   - eg handling user input, managing state, defining methods, etc.
   - similar to the Model piece of the MVC design pattern (RECALL COMP301)
2) An HTML template that defines & determines the user interface
   - controls what is rendered to the browser.
   - Similar to View, from MVC
3) Component-specific styles, written in CSS pages, that define the "look & feel"

What is a component class' job (ideally)?
→ to enable the user experience and nothing more! A component should:
- present properties (class fields) and methods for data binding
- use these fields & methods to mediate between the View (the html template) and the application logic (which usually includes some notion of a **Model**)

→ Angular does not enforce these principles (of what a component should & shouldn't do) — this is just like, best practice (similar to "principles of encapsulation" in OOP, RECALL COMP301)

Crash Course on Widgets in Angular

What are "widgets" in angular?
Why are they useful?

→ individual, reusable user interface elements that can be easily integrated into the UI of your Angular components! (aka "component.ts" files).

→ makes your Angular frontend more versatile & modular.

→ widgets basically "abstract" frontend UI elements in order to simplify Angular components.

* makes the development process in Angular less painful.

– Review of Angular Modules –

→ Angular Modules define the application's **structure** & help to manage dependencies in an Angular application

→ A module in an Angular proj organizes & encapsulates components, services, and other code that is related to a specific feature/functionality.

- like the app.module.ts class in EX00

How many modules should a proj have?

→ Not just one – this results in longer load times & stuff because, when you load your app onto the browser, the entire module & all of its declarations have to be loaded.

→ Its better to encapsulate features into separate modules.

→ Ex: The CSXL Web app has separate modules for each subpage – organization.module.ts, event.module.ts, etc.

- but it still has an app.module.ts to ultimately encapsulate the app.

So where should widgets be stored?

→ To understand this, lets consider 3 example widgets:

- A <search-bar> widget that defines a search bar item to be used **throughout the application.**
- A <social-media-icon> widget that defines social media icon buttons to be used **throughout the application.**
- A <organization-card> widget that provides the format for the little boxes which display info about CS orgs at UNC, to be used on "**organization**" **pages**.

→ widgets that will only be used for one page, like organization-card, can be stored in that page's module.ts file.

→ For "global widgets", like the other two, should be declared in a separate "shared.module.ts" file, that the other modules that need them can then import.

How do you declare a new widget?

→ In a /widget folder (which should be either in an individual module's folder or in the "shared" folder), create a new folder titled <widget-name>, & create the following 3 files inside:

1. widget-name.widget.css
2. widget-name.widget.html
3. widget-name.widget.ts

(similar to creating components!)

How do we define the widget's behavior?

→ In the .ts file, starting with this template:

the "@component decorator"

```
@Component ({
    selector: 'widget-name',
    templateUrl: './widget-name.widget.html',
    styleUrls: ['./widget-name.widget.css']
})

export class WidgetName {
    /* Inputs and outputs go here */
    /* Constructor */
    constructor () { }      }
```

the selector property, which is the name that we will use when referring to the widget in the HTML file.

→ Once we create this template, we must declare it in a module!

→ In the ___.module.ts file of the folder that the widget was made in (like organization.module.ts or shared.module.ts, for ex):

- add the name of the widget class created to the list for the declarations property in the @NgModule decorator.

How do you pass data into a widget?

→ Using the @Input decorator in the WidgetName class fields:

```
export class OrganizationCard {
    @Input() organization!: Organization
    constructor () { }
}
```

the "!" unwrap operator ensures to TypeScript that the organization field is required & will be passed into it upon construction of an OrganizationCard object.

the required input of this field is an Organization object.

→ With the straight brackets [ ], which denote inputs (in Angular)

How do we display this widget on a page?

→ Inside the html file of the desired component that we want to add the widget to:

organization-page.component.html:

```
<organization-card [organization]="organization"/>
```

- This adds one organization card widget to the display of the organization page. It declares 1 OrganizationCard object & passes a specific instance of an Organization object (aka some CS org @ UNC) into the input, using [ ].

How can we display one organization-card widget for each organization?

→ Using the "*ngFor" structural directive, which renders/dynamically repeats a template for each item in a collection:

```
<organization-card [organization]="organization" *ngFor="let organization of organizations"/>
```

How do we send data from widgets to components?

→ AKA, if we have a button on our widget, how do we code an action into that button? i.e., how can we get the button to trigger a function in the parent component?

- the answer: using the @Output decorator!

What does the @Output() decorator do?

→ allows us to pass data from widgets back to components — the opposite of what we did with @Input().

- then, the parent component can run functions using the output of the widget.

RECALL: how do we define a basic button in Angular?

→ In the html file of the respective component:

```
<button (click) = "myAction ()" > Click me lol! </button>
```

↓ the text that would appear on the button.

→ the parentheses after a UI element is syntax for an **event binding**.

→ This example button basically says that whenever the button is clicked, the function which has been passed into (click) is run — aka the myAction() function.

- the (click) event & myAction() function are now bound together.

Example — how can we add a button to the organization-card widget that allows the user to "join the organization"?

→ Several Steps:

1. define a function in the widget's parent component class — in this case, organization-page.component.ts — that actually performs the action/code of 'joining a user to an organization'
   - SIMILAR: to writing methods in the Model class (MVC)

organization-page.component.ts

```
joinOrganization (org: Organization) {
    // implementation code here }
```

2. Define an output field in the widget class/TS file — in this case, organization-card.widget.ts
   - → remember, we have many org cards (one for each organization), and each card will have this "join" button
   - → So, we need the output so that there is a way for the organization-page.component class to know which organization to join when button is pressed.

(Example code on next page)

organization-card.widget.ts

```
export class OrganizationCard {
  @Input() organization!: Organization

  @Output() joinButtonPressed = new EventEmitter<Organization>()

  constructor () { }
}
```

↳ an "event handler" for when the button is pressed.

3. Connect the event handler defined in the widget TS file, **to the html of the parent component!**

→ Now that we've defined this event handler, we can access it in instances of the widget that we created in the parent components html file (in this case, organization-page.component.html)

BEFORE:

```
<organization-card [organization]="organization" />
  *ngFor ="let organization of organizations"
```

NOW:

```
< organization-card
    [organization] = "organization"
    (joinButtonPressed) = "joinOrganization(org: $event)"
   *ngFor = "let organization of organizations"
     />
```

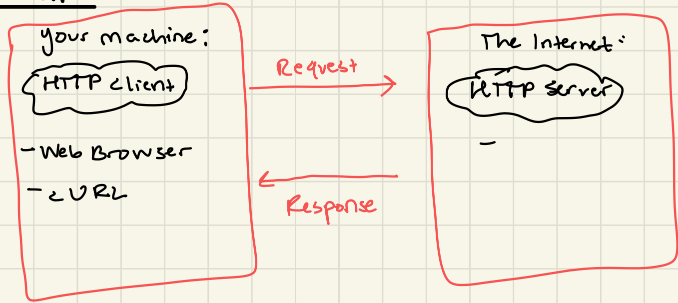
This line accesses the output of the joinButtonPressed event handler

→ Didn't finish notes bc its confusing & pointless

Pull Requests

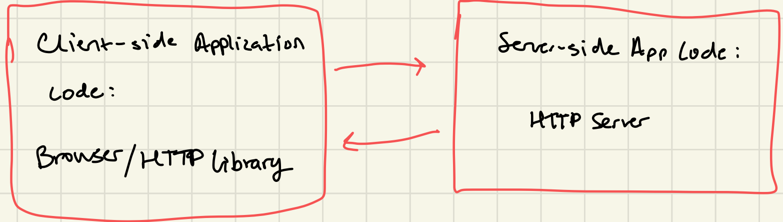
What is a merge?	→ A way to bring changes from one branch to another.
What is the typical work flow for larger teams?	1. create branch WIP off of main 2. create branch feature off of WIP 3.
Merge Conflict:	→ When branches have overlapping changes to the same file(s). → Merge Commits: created whenever you merge branches with <u>divergent</u> histories ... <u>not</u> the same as merge conflict
What is a pull request?	→ can be better thought of as a "merge request" & acts as an official record of a merge between branches. → A PR includes an area for the PR's creator to leave comments, and f other collaborator's to do the same.
Why use one?	→ allow for easier code reviews and catching bugs. → Documenting changes for your team & others → Alerting other collaborators that a branch has been updated → Is shared among your team, not just on your local machine → Simplified workflows : Lets you make a merge with the press of a button!
	→ Fast forward: when a merge occurs w/o a merge commit.

Client ↔ Server Interaction



→ All communication happens over "requests" & "responses"

→ Usually begins w/ a request from the client.



Understanding Dependency Injection in Angular

What is Dependency Injection?

→ The mechanism that manages the dependencies of an app's components, and the services that other components can use.
→ DI is one of the fundamental concepts in Angular.
→ Essential to the Angular decorators, such as Components.

What are the 2 main roles in the Angular DI system?

1. The dependency provider
2. The dependency consumer

→ Angular facilitates the interaction between the two using an abstraction called Injector

What does Injector do?

→ When a dependency is requested, the injector checks its registry to see if there is an instance already available there. If not, a new instance is created & stored in the registry.

How do you enable a class to be 'injectable'?

→ By adding the @Injectable decorator to the top of the class file.
→ Example: A class called Book that needs to act as a dependency in a component;

```
@Injectable()
class Book() {
    ... }
```

How do we provide the DI of a class to a component?

→ By adding the class to the list of providers — which is a field of the @Component decorator;
→ The injected class becomes available to all instances of the component, as well as all other components & directives used in the template!

homepage.component.ts

```
@Component({
    selector: 'homepage',
    templateURL: './homepage.component.html',
    StyleURLs: ['./homepage.component.css'
    providers: [Book]
    })

export class HomepageComponent() {
    ... }
```

RECALL: the selector property defines the name used when referring to a component in its html file.

homepage.component.html also gains access to the Book class objects

Recall: these 2 fields refer to the other 2 of 3 files that are created when you "ng generate component" — an html & a css file.

What does DI at the component-level do?

→ When you register a provider at the component level, you get a new instance of the "service" (in this case, a new instance of the Book class object) with each new instance of the component.

What if we want to provide a single, shared instance of a class for DI?

→ We can do this by adding the `providedIn: 'root'` field to the decorator of the class being injected:

```
@Injectable({
    providedIn: 'root'
  })
  class Book() {
      ... }
```

→ When we provide the class at the root level, Angular creates a singular, shared instance of Book and injects it into any class that asks for it.

How do we provide the DI of a class to an entire application?

→ Provide it inside of the entire app's AppModule – aka app.module.ts !

→ Add the class to the list of providers in the providers field of the class' @NgModule decorator:

app.module.ts :

```
@NgModule({
  declarations: [
      ... ],
  imports: [
    ... ],
   providers: [
    { provide: Book },
  ] };
```

What does DI at the application-level do?

→ When you register a class as a provider at the application level, the same instance of the class/service (Book) becomes available to all components, directives, and pipes declared in this NgModule.

→ difference between root-level & application-level DI ?

So how do you actually inject a dependency, once you have set up the "providing" aspect?

→ There are 2 ways to do it:

1. Constructor Injection – declare the service in a class constructor
   - RECALL: COMP 301 DI & constructor injection notes

   Ex:

   homepage.component.ts

```
@Component({...})
class HomepageComponent {
    constructor (private service: Book) {}
  }
```

2. Using the inject method:

homepage.component.ts

```
@Component({...})
class HomepageComponent {
    private service = inject(Book);
}
```

What happens when Angular sees that a component depends on a service?

→ it first checks if the injector has any existing instances of that service.

→ If a requested service instance doesn't exist yet, the 'injector' creates one using the registered provider, & adds it to the injector before returning the service to Angular.

Models in Angular

What do we use interfaces in TypeScript?

→ To represent/model a collection of data

→ We can declare an interface directly inside of another class. For ex:

organization-page-component.ts

```
export interface Organization {
    name: string;
    description: string;
    yearFounded: number;
    events: string[];            }
```

→ This defines a sort of "organization" object type that has all the properties

→ The export interface allows the interface to be used outside of the .ts file where it was declared.

Why do we use interfaces in this way?

→ The interface just specifies the "shape" of an object – aka the properties that we expect

- as opposed to creating a class, where we'd be required to have a constructor, methods, etc.

→ Basically just a way to group certain properties together which define an object (like an 'organization').

Why does this approach work in T.S. but not Java?

→ Because of **structural typing**! We can create an object of an <u>interface type</u> (something you can't do in Java!)

→ For example:

```
let csxl : Organization = {
    name: "Experience Labs",
    description: "cool!",
    yearFounded: 2023,
    events: ["workshop 1"]
}
```

Here, we have used an object literal to create a new obj of type "Organization"

What is an object literal?

→ The example above -- a way to spontaneously construct an interface object without a constructor or new keyword or anything.

- is a way to declare a variable

→ This approach to using an interface is called **a model**.

Services in Angular

What is a service?

→ A broad category encompassing any value, function, or feature that an application needs.

→ Typically a class with a narrow, well-defined purpose

→ Completely separate from components -- created using

`ng generate service <servicename>`

What is the purpose of Angular services?

→ They provide a way to separate Angular app data, & functions that can be used by multiple components in your app.

→ A centralized data source that multiple components can use.

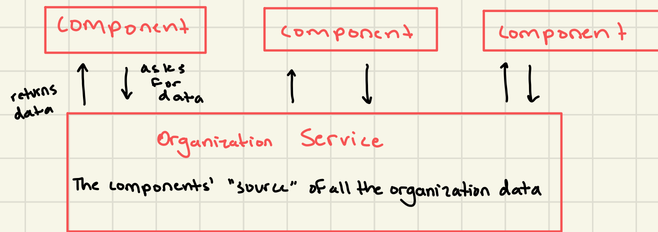
When should a component use a service class?

→ For tasks that don't involve the View or the application logic. For ex;

- fetching data from the server
- validating user input
- logging directly to the console (`console.log("...")`)

→ These processing tasks are the type of things that we should then define in a service class!

- By defining them in an injectable service class, we make the tasks available to any component.



How do you make a service usable by multiple components?

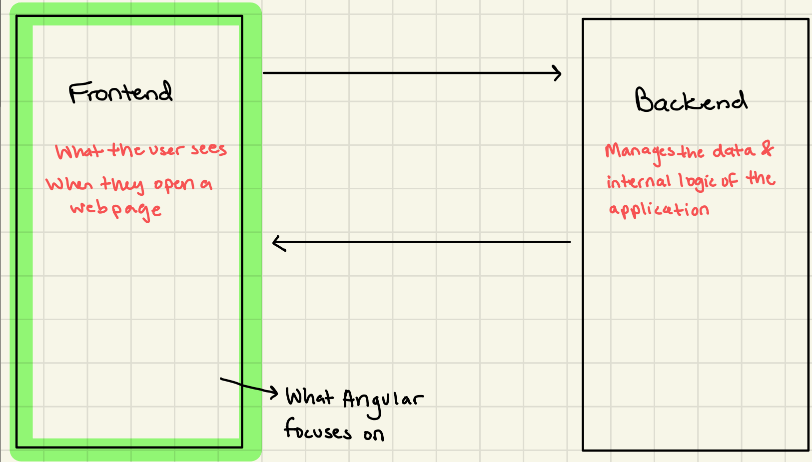
→ The service must be made injectable - by adding the @Injectable decorater to the service class!

→ services that are injectable & used by a component become dependencies of that component -- that is, the component can't function w/o them.

Intro to Angular videos

What is the structure of a web application?

→ 2 parts: the **Frontend** ("client side") and the **Backend** ("server side")

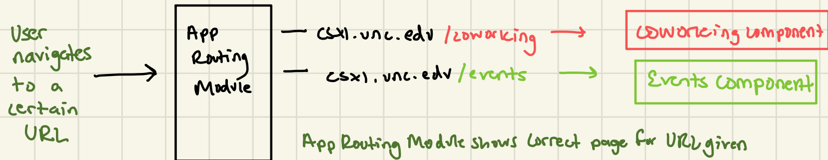


→ Angular is used to make **single-page web applications**.

- Application made up of a collection of views
- when user navigates to one of the URLs, the application **dynamically rewrites the page to show the correct view.**

→ Each component is a "view" associated with some URL.

What is the "Angular App Routing Module"?



→ responsible for showing the correct component based on the URL that the user navigates to.

Quiz Review

T.S. Syntax

→ Typescript utilizes structural typing -- views objects as being of the same type if they share the same structure
- regardless of if they have the same type name ("nominal typing")

→ `const` in T.S. = "final" in Java

→
```
type Rating = number;
let csxL : Rating = 10;
```
```
type Animal = Cat | Dog;
```

Type annotations

→ type annotations are a way to explicitly specify the expected data type of a parameter, return val, or any other variable being declared in a program.
- In TypeScript, type annotations are optional (RECALL: the code from EX00) because TS is usually able to infer the expected data type.
- However, they are useful to the compiler in checking types, and can help avoid errors dealing w/ data types

Higher Order Functions

1. having a function be the return type of another function
2. having a function be a parameter arg of another function

HTML and TS File relationship in a Component

→ Property Binding:
- enables HTML element to use values from TS file as a **property input**
- aka, setting the value of one of the properties (~class fields) of the TS file inside of the HTML file.
- Stuff inside of quotation marks is TypeScript language code.

```
[price] = "product.price"
```

- `[ ]` denotes an input

→ Event Binding:
- enables HTML element to call functions from the TS file.

```
(buyButtonPressed) = "purchase(product)"
```

- `( )` denotes an event binding
- quotations indicate TS code, where the function is called.

→ Interpolation Syntax:
- enables the use of a value/field from the TS file to be inserted directly into the HTML file

```
<mat-card-title
    {{ organization.name }}
</mat-card-title>
```

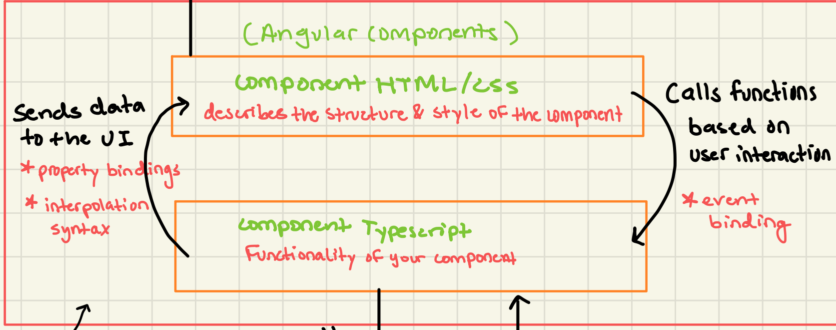
- the TS value is placed into the UI (via the HTML & `{{ }}`) as text

User navigates to a URL
- which is provided to App Routing Module

Active view
- that the user sees & interacts with

Connects components to URL paths
- so that they can be accessed from browser.

Angular App Routing Module



Components expose a **static route**
- describes the URL that the component is available at

can also be used to represent data in the components as well

components **call** service obj.s' functions
- to retrieve data
- to take care of non-UI functionality

services send data
- for components to display in the UI
- services are **injected** into components

Data Models
- the interface things w/ only properties (no methods)

Define the structure
of data that the service expects to work with

Angular Services
- centralized data source for multiple components
- "source"
- made available to components through DI

FastAPI

Question	Notes
What is an API?	→ API = Application Programming Interface → an API is a software / "set of programming code" that sort of functions as a set of rules / protocols that let different computer programs and/or software applications communicate with each other to exchange data, features, and functionality. • Like the backend with the frontend!
What is FastAPI?	→ A leading, modern framework for developing backend APIs in Python → a backend server framework.
What is a backend server?	→ A server running in our cloud that the client-side of an application can send requests to to do things.
What do we use FastAPI for?	→ To implement server-side functionality. • When your server is running, it is within the container → You can make requests to the server using your web browser, or other tools – such as `curl`. → The functions we implement on the backend are less trivial, and more focused on one objective / responsibility: persisting or retrieving data to and from a database.
What is the command to run the live server of an API?	→ `uvicorn main:app --reload`, where • `main` refers to the Python file that contains the `FastAPI()` object. • `app` is the FastAPI object created inside `main.py` with the line `app = FastAPI()` • `--reload` is a command that makes the server restart after code changes.
What is a schema?	→ a definition or description of something — Not the code to implement anything, but just an abstract description.
What is a data schema?	→ A "schema" that refers to the shape of some data, like a JSON content. • in that case, it would include the JSON attributes, the data types they have, etc.
What is an API "schema"?	→ OpenAPI is a specification that dictates how to define a schema of your API. • it defines an API schema for your API. (?)
What does the OpenAPI schema definition include?	→ your API paths, the possible parameters they take, etc. → it can also include definitions – aka other schemas – of the data sent and received by your API using JSON Schema – the standardized format for JSON data schemas.

What does FastAPI do with OpenAPI schemas?

→ FastAPI generates a "schema" with all of your API using the OpenAPI standard for defining APIs

What is OpenAPI for?

→ The OpenAPI schema is what powers the interactive API documentation systems (the website that shows all the parts of your API)

→ It can also be used to generate code automatically for clients that communicate with your API.

- i.e., for frontend or mobile applications.

How do you create an API in Python using FastAPI?

→ In a new Python file, we do the following steps:

1. import FastAPI — a Python class that provides all the functionality for your API:

```
from fastapi import FastAPI
```

2. Create a FastAPI object "instance". This object will be the main point of interaction to create all your API.

```
app = FastAPI()
```

3. Create a path operation

What is a "path"?

→ The last part of a URL, starting from the first / .

- For ex, the "path" of https://example.com/items/foo would be "/items/foo"

→ When building an API, the path is the main way to separate "concerns" and "resources".

What is an "operation"?

→ Refers to one of the HTTP "methods", such as POST, GET, PUT, DELETE, OPTIONS, HEAD, PATCH, and TRACE.

How do we use operations when building APIs?

→ in HTTP protocol, these methods are used to communicate to each path.

→ When building APIs, we normally use a specific few HTTP methods to perform specific actions:

- POST : to create data
- GET : to read/retrieve data
- PUT : to update data
- DELETE : to delete data

→ In OpenAPI, each of these methods are called operations.

So how do we create a path operation (step 3)?

→ With a line in the format @app.<operation>("<path>")

```
@app.get("/")
//function code here
```

• This line tells FastAPI that the function code below it is in charge of handling requests that go to the path "/" using the get operation!

What is "@app"?

→ We can do this with any of the operations; @app.post(), @app.put(), @app.delete(), etc.

→ The path operation decorator

→ A decorator (@something) in Python is something that you put on top of a function if you want to do something specific with it.

- @app is a decorator that tells FastAPI that a certain path corresponds to a certain operation.

4. Define the path operation function — the function below the @app decorator line.

- For a path operation @app.operation("/path"), the Python function under it is what will be called by FastAPI whenever it receives a request to the URL "/path" using an 'operation' operation.

```
@app.get("/")
async def root():
    ...
```

→ the "async" function named root, which will be called by FastAPI

• we could also have excluded the "async" keyword and made this a normal function.

5. Return the content!

All the code all together:

main.py

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/")
async def root();
    return {"message": "Hello World"}
```

→ There are many objects and models that we can return & that will be automatically converted to JSON.

- Just a few possible things we can return: a dict, list, str, int, etc.

What will this API display when we run the live server?

```
from fastapi import FastAPI
app= FastAPI()
@app.get("/")
async def root();
    return {"message":"Hello World"}
```

On localhost:8000, the following page:

```
{
  "message": "Hello World"
}
```

FastAPI: Path Parameters

What are path parameters?

→ declaring path "parameters" basically means setting a part of the "path" argument that goes into a path operation, to be some valve that can then be passed into the path operation function as an argument, where it can then be stored/displayed/otherwise used.

Example?

→ Compare this example from the previous page:

```
@app.get("/")
```

to

```
@app.get("/items/{item_id}")
async def read_item(item_id):
    return {"item ID": item_id}
```

→ the { } syntax indicates that this portion of the website path should be declared as a variable/parameter called item_id

The path operation function takes the path in as a parameter, and then displays it by calling the variable name, with no quotes around it.

What will this API display?

→ Whatever we type into the address bar for the path will then get displayed.

```
127.0.0.1:8000/items/foo
{
  "item ID" : "foo"
}
```

→ If we change the path, the page updates!

```
127.0.0.1:8000/items/ella
{
  "item ID" : "ella"
}
```

How can we declare the data type of a path parameter?

→ We can specify the data type inside the path op. function, using standard Python type annotations

→ By specifying a type, we prohibit the path from being anything else. If we change the path to an incorrect type, an HTTP error message displays.

Example?

```
@app.get("/items/{item_id}")
async def read_item(item_id: int):
    return {"item ID": item_id}
```

→ here, item_id is declared to be an int

The browser →

```
127.0.0.1:8000/items/3
{
  "item ID" : 3
}
```

→ But if we change the path to 127.0.0.1:8000/items/foo :

```
{
  "detail": [
    {
      "type": "int_parsing",
      "loc": [
        "path",
        "item_id"
      ],
      "msg": "Input should be a valid integer, unable to parse string as an integer",
      "input": "foo",
      "url": "https://errors.pydantic.dev/2.1/v/int_parsing"
    }
  ]
}
```

} error message

What if we want to use a path parameter, but also have a fixed path?

→ We can do this by declaring multiple path operations. However, we want to declare the fixed path first, because path operations are evaluated in order.

Why does path operation order matter?

→ Because when the code is read/compiled(?), the computer only executes the function under the first path operation it sees that matches the current browser path.

- This is the same reason that you cannot "redefine" a path operation;

```
app = FastAPI()
@app.get("/users")
async def read_users():
    return ["Rick", "Morty"]

@app.get("/users")
async def read_users2():
    return ["Rick", "Morty"]
```

when you navigate to the page 127.0.0.1:8000/items/users, the first function is the only one that will ever display since the path matched first.

Example of using multiple path operations?

```
app = FastAPI()
@app.get("/users/me")
async def read_me():
    return {"user_id": "the current user!"}

@app.get("/users/{user_id}")
async def read_users(user_id: str):
    return {"user_id": user_id}
```

→ We have to declare the fixed path function for the path "me" first, otherwise the computer would match the path to the read_users function, assuming that "me" is just the path parameter arg for user_id.

What if we want the possible valid path parameters to be predefined?

→ We can use a standard Python Enum class!

1. Import Enum and then create an Enum sub-class inside of your "main.py" API file
   - Make the class inherit from Enum **as well as from str** — so that the API docs know that the values must be of type String and will be able to render them correctly.
2. Create a **path parameter** with a <u>type annotation</u> of the name of the enum class
3. Since the value of the path parameter arg will be an **enumeration member** and can't be anything else, we can use "is" to compare the param. input to our enumeration members
4. Alternatively, we can just directly get the value of the input by using <your_enum_number> .value
5. Our path operation can even return **enum members** — they will be automatically converted to their corresponding values (in this case, strings) before being displayed.

```
from enum import Enum
from fastapi import FastAPI

1. class NetNames(str, Enum):
       alex = "alexnet"   ]
       res = "resnet"     }  all of the available, valid values
       len = "lenet"      ]

app = FastAPI()

                                              the
@app.get("/models/{modName}")
2. async def get_model(modName: NetNames):
   3. if modName is NetNames.alex:
          return {"model_name": modName, "message": "Learn FTM"}

   4. if modName.value == "lenet":
   5.     return {"model_name": modName, "message": "Goodbye!"}
```

What will the client browser look like?

127.0.0.1:8000/items/alexnet:

```
{ "model_name" : "alexnet",
  "message" : "Learn FTM"
}
```

Can we have a path parameter that contains a path itself?

→ e.g., a path operation with a path /files/{file_path} where the parameter is a path itself, like file_path = home/janedoe/myfile.txt
- so the URL would, in full, be /files/home/janedoe/myfile.txt

→ OpenAPI doesn't support a way to declare a path parameter to contain a path inside
- However, we can still do it in FastAPI, using one of the internal tools from Starlette, the class that FastAPI inherits directly from.
- the open API docs would still work

How do we implement this, using Starlette?

→ Using the ":path" annotation on the parameter variable.

→ For example:

```python
from fastapi import FastAPI
app = FastAPI()

@app.get("/files/{file_path:path}")
async def read_file(file_path: str):
    return {"File Path": file_path}
```

declares that everything in the URL after /files/ will be saved as a param variable of type path

The client browser:

localhost:8000/files/avikumar/lol:

```
{ "File Path" : "avikumar/lol"
}
```

Recap: what is the advantage of using FastAPI for path parameters?

→ by using short & intuitive Python type declarations, we get:
- Editor support – error checks, autocompletion, etc.
- Data "parsing"
- Data validation
- API annotation & automatic documentation.
- we only have to declare them once.

FastAPI : Query Parameters

What are "query parameters"?

→ Any other parameters that we declare in the def ______ (path operation) function that aren't path parameters are interpreted to be query parameters.

- RECALL: we know if/when it is a path parameter because the path in .get("/...") will contain a portion enclosed in curly brackets, indicating that it is a parameter that the function will then use

  @app.get("/items/{item_id}")

What is a query?

→ in a URL, the query is the set of key-value pairs that go after the ? in a URL. The key-value pairs are separated by & characters

→ EX in the URL http://127.0.0.1:8000/items/?skip=0&limit=10, the query parameters are:

- skip, with a value of 0
- limit, with a value of 10

Example of declaring query parameters?

fig.1

```python
from fastapi import FastAPI

app = FastAPI()

@app.get("/items/")
async def read_item(skip: int, limit: int):   # 4
    return {"skip num is": skip, "limit num is": limit}
```

(type declaration → int)

- the path op. function takes in 2 parameters, an int "skip" and an int "limit"
- Since they are assumed to be query parameters, the executer will look for these param args in the URL after the ? character.

What's the point of declaring skip and limit as type int?

→ Since query params are part of the URL, they are "naturally" strings.

→ But by declaring them with Python types, the params are converted to that type & validated against it.

- This is good; it gives us the same advantages as those described for path parameters (on prev. page)

Can query parameters have default values?

→ Yes! Since the query stuff isn't a fixed part of a URL path, they can have default values.

→ Give query params default values by declaring them inside the function's argument:

```python
async def read_item(skip: int = 0, limit: int = 10):
```

How would this present in the client browser?

→ if we replace line 4 in fig.1 with the line above, then going to the URL localhost:8000/items/ (aka no query params) displays this ↓ because we set skip and limit to 0 and 10 by default.

```
{"skip": 0;
 "limit": 10; }
```

→ The URL localhost:8000/items/?skip=0&limit=10 would effectively be the exact same, and also display this ↗

How can we declare query parameters as optional?

How do we use bool (boolean) values as query parameters?

What does the "|" symbol do?

What is the "item" object?

What does "item.update" do?

→ localhost:8000/items/?skip=15&limit=5

{"skip": 15, "limit": 5}

→ localhost:8000/items/?skip=15&limit=aabb would produce/display an http error since it isn't of type int.

→ By setting their default values to be "None"!

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/items/{item_id}")
async def read_item(item_id: str, q: str | None = None):

    if q:
        return {"item ID is": item_id, "q is": q}
    return {"item ID is": item_id}
```

if nothing is provided for q, it is set to "None" by default

if q: → aka, if q ≠ None

→ used for 'type hinting' to indicate that q is a Union type, meaning that it can be any of the specified types

→ Used if you want a parameter to accept multiple data types.

→ For ex, var: str | None means that var can either be a string, or a "None" type.

→ We can also just easily make query parameters required/non-optional by not declaring any default value.

→ Same as any other d.t., for ex:

```
from fastapi import FastAPI
app = FastAPI()
@app.get("/items/")
async def read_item(short: bool = False):
    item = {"message": "hello!"}
    if not short:
        item.update({"description": "I love food"})
    return item
```

setting default value of short to false

if short is false, then item is updated.

→ NEW: notice how we can create an object ("item") that stores the entire JSON response body... which we can then return in the path op. function! As opposed to directly creating the response in the return statement.

→ A way to update/add contents to the JSON response stored in item.

→ In the example above, the description field is added on after the item_id field.

How do we pass a bool query param into the URL?

→ Any of the following would be accepted for the parameter short defined in the example:

interpreted as short = TRUE	interpreted as short = FALSE
localhost:8000/items/?short=1	localhost:8000/items/?short=0
localhost:8000/items/?short=True	localhost:8000/items/?short=off
localhost:8000/items/?short=true	(and so on)
localhost:8000/items/?short=on	
localhost:8000/items/?short=yes	

→ And even more, b/c its not case sensitive.

FastAPI: Request Body

What is a request body?

→ data sent by the client (like a browser) to your API

→ in FastAPI, we declare request bodies using Pydantic models

- Pydantic: a Python library for data modeling and parsing that has efficient data & error validation.

What operation do we use to send data from the client?

→ One of post(), put(), delete(), or patch() — post() is the most common.

→ Almost never use get() to send data.

How do we declare a request body?

1. Create a data model that defines all of the fields of the request body, and the data type(s) that they should accept.
   - We declare this data model as a class - specifically, a class that inherits from Pydantic's BaseModel class - we will have to import pydantic BaseModel to do this.
   - Once filled in with data from the client, this model will be used to declare a JSON "object" (like all the browser outputs we've seen so far) that displays all of the fields.

   → For example, this model could declare a JSON object like

```
{ "name": "Foo",
  "description": "hello",
  "price": 45.2 }
```

2. Create a .post() path operation whose function takes the data model class object as a "body" parameter.

```
from fastapi import FastAPI
from pydantic import BaseModel


class Item(BaseModel):              # Step 1.
    name: str
    description: str
    price: float | None = None      # we can give model attribute's default values the same way as with query parameters.


app = FastAPI()
@app.post("/items/")                # Step 2.
async def create_item(item: Item):
    return item
```

Can we declare multiple types of parameters at the same time?

→ Yes! We can declare **body**, **path**, and **query** parameters all in the same function, and FastAPI will be able to recognize which is which & then take data from the correct place.

```
@app.put("/items/{item_id}")
async def updateItem(item_id: int, item: Item, query: str = "hi"):
        ....
        return ...
```

How are the function parameters recognized?

→ if the parameter is also **declared** in the path, it will be used as a path parameter.

→ if the parameter is of a **singular type** - like int, float, bool, str, etc. - it will be interpreted as a query parameter.

→ if the parameter is declared to be the type of a **Pydantic Model**, it will be interpreted as a **request body**.

Lecture Notes – Observables and HTTP Client

What do we specify when making an HTTP request (from the frontend)?

→ A few things:

1. The action that we want to perform – aka one of the HTTP methods (such as GET, POST, PUT, and DELETE)
2. The API endpoint – refers to the URL where your API is available
   - for example, the API for the csxl organization page is at the endpoint csxl.unc.edu/api/organizations
   - THIS URL is where all of the JSON schemas with the data are displayed (like the examples from the FastAPI tutorial!)
3. (occasionally, not always) a request body: data that you are sending along your request

Where does the frontend connect to the backend?

→ Angular services! The same ones we've been using

- The service class is where we make the HTTP requests
- service classes are responsible for calling the correct backend APIs

What is the "HTTP Client"?

→ A thing built into Angular that allows the service class to make HTTP requests & receive HTTP responses

- inject the HTTPClient object into the service class' constructor

Example in an Angular Service class?

```
export class AssignmentService {
    constructor (protected http: HTTPClient) {}
    getData () {
        this.http.get<T> (/api/organizations) }
```

this is the 'API endpoint'

(example doesn't include params or return type... keep reading!)

What are the HTTPClient methods?

→ They correspond by name to the HTTP methods:

- `http.get <T> ( API endpoint );`
- `http.post <T> ( API endpoint , requestModel);`
- `http.put <T> ( API endpoint , requestModel);`
- `http.delete <T> (API endpoint/ + id );`

the data that we want to send to the backend – in the form of a data model (aka TypeScript interface)

the type, T, is the data model that API works with -- aka a TypeScript interface!

What is the return type of these methods?

→ an Observable <T> object, where T is the data model!

What is an Observable?

→ Part of the RxJS package

→ represents an asynchronous stream of data

→ To access a value from an observable, we subscribe to it.

Example of subscribing to an Observable?

→
```
myObservable.subscribe((nextValue) => {
    console.log(nextValue);
});
```

→ the .subscribe() method takes a function as a parameter

→ Whenever/every time that a new value/data appears in the stream, it is populated into the parameter input of .subscribe - in this case, the nextValue field.

- then, the function that we provided will be run (every time the value is updated)

So how is this observable used in the Angular Service (with HTTPClient)?

→ Revised example of an Angular Service class:

```
export class AssignmentService {
    constructor (protected http: HTTPClient) {}
    getData(): Observable<Assignment[]> {

        return this.http.get<Assignment[]>('/api/assignment');
    }
```

specified return type of the method, where Assignment is the data model that API works with.

making an HTTP request to the backend API

Explanation of how this code connects to the API?

→ RECALL: the FastAPI Python file contains a bunch of **path operations** (labeled by the @<api name> decorator), each of which specify 2 things:

1. An operation - i.e. .get, .put, .post, etc.
2. The path associated with the operation function

→ Then, under each of these path operations is a **path operation function**, which is essentially a method that performs some task regarding the data (which it receives from the backend service)

→ for Ex, the assignment.py API file for the above example might look like:

```
api = APIRouter(prefix = "/api/assignment")*
@api.get("")
    def get_assignment(backend service):    # parameter from backend service
        return backendService.assignmentList()
```

- the above method returns an Assignment[] list through some method from the backend service.

→ so BASICALLY, if we want to execute the task/code of one of the api operations from the frontend (service class), we use the HTTPClient method and pass in the "path" of the operation which we want to execute & use!

- In this example, calling this.http.get<Assignment[]>('/api/assignment'); successfully routes to the APIs get_assignment function because we passed in this endpoint, and .get("") refers to the Router endpoint prefix*, so its a match!

How do you use the object returned by the HTTPClient methods?

→ Since the returned object is an Observable <data model>, we can't just directly call the method to return a data model object.

→ Instead, we call the method which returns an observable, & then subscribe to it & provide the desired code inside the function that .subscribe() takes as a parameter.

Example?

→
```
export class AssignmentService {
  constructor (protected http: HTTPClient) {}
  getData(): Observable < Assignment[] > {
    return this.http.get< Assignment[] >('/api/assignment');
  }
```

In another frontend class:

```
myAssignmentService.getData().subscribe((assignments) => {
    console.log(assignments);
});
```

(to print the list of Assignment objects)

Why would the HTTP client want to use Observables instead of returning the value directly?

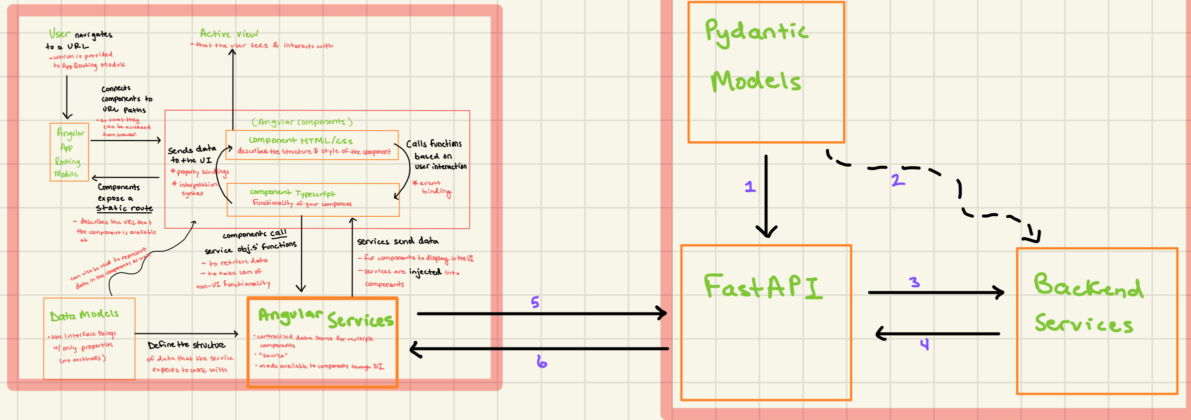
→ if we waited on the server directly to process what it needed to do, it would be synchronous and we would **block** on the function call — meaning our program would stall and become choppy

→ Basically, we use Observables for **asynchrony**.

Tech Stack Diagram

The Frontend

The Backend



1. Pydantic models **define the structure** of data that is sent out by the backend APIs
2. They also define the structure of data that FastAPI expects to work with.
3. FastAPI calls **service functions** to perform **CRUD** (create, read, update, delete) operations on the backend data. (services are **injected** into FastAPI)
4. Services send a response (either data, a message, or an error) based on the type of CRUD operation that was performed.
   - This response will ultimately be sent to the frontend across HTTP
5. HTTP Request (aka API CALL): services connect to the backend by **calling APIs** using an **HTTP Request** (see lecture notes)
   - the specific <u>request path</u> and <u>type</u> (HTTP method) determine the type of data received in the HTTP Response
6. HTTP Response: Depending on the FastAPI path and request type, FastAPI **will reply** with a response.
   - response either **contains the data requested**, or it may be an error with an HTTP error code.
   - No matter the type, the Angular service must be able to respond to any type of request.

→ our API is made up of a bunch of routes that are a combination of one of the HTTP methods, & a URL extension (after /api)

→ main.py is the top level api file.

→ APIs, like Angular components, also have their own routers for URLs
   - in main.py, you can define a list of other API python classes that contain API requests & other code thats more specific to individual parts of the csxl website

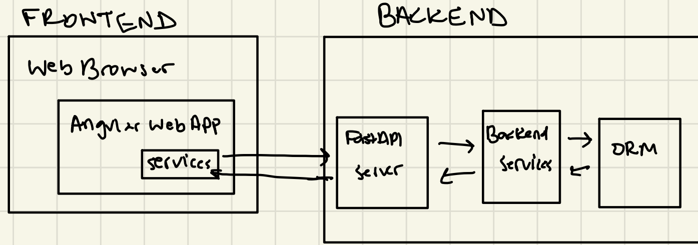
→ Definition/notion of "**the backend**":
   - we define some functions
   - we add an annotation to those functions specifying the **route** and the **method** that we want them to respond to.
   - we implement it and (typically) back it up by a service.

Quiz Review

- → PUT : sends data from the **frontend TO the API**
  (to "create" data, when used by fastAPI)
- → GET : sends data from the **server (aka API) TO the webbrowser (aka frontend)**... to **retrieve** data (when used by fastAPI)
- → PUT : to update data
- → HTTP header fields : a list of strings sent & received by both the client program & the server on every HTTP request & response.
  - processed & logged by the server & client applications (not visible to end-user)
  - possible fields: Content-Type, Accepts, Content-Length,
- → Status codes are included in **HTTP Responses** (client → server)
- → HTTP status code meanings:
  - 400-level -- issue on client-side
  - 300-level -- redirection message
  - 500-level -- Internal server error
- → Inject backend service into API function:

```
def Function(service: Service = Depends()):
```

Class 2/26

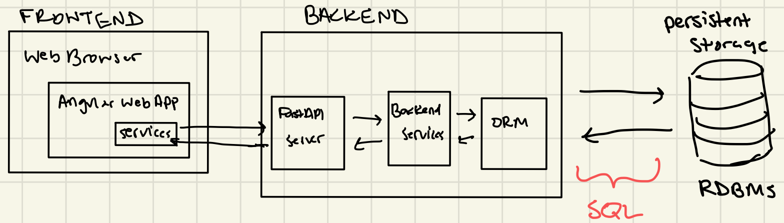


→ Data doesn't save when we restart our Python/Cloud server

→ we need a third tier: persistent storage

→ persistent storage

RDBMS



What is RDBMS?

→ Where data lives (?)

→ Relational database management system

What is SQL?

→ Structured Query Language – a declarative language

; just used to specify what you want (and make "queries")

What is the ORM layer?

→ ORM = Object Relational Mapper

→ "Sql Alchemy" is an ORM

→ the RDBMS is what lets us store long-lived storage for our application

- storing it here rather than in the frontend or backend brings powerful capabilities.

→ RDBMS is a separate component from frontend or backend; it is postgress

→ RDBMS gives us Transactions which have ACID properties

A – ATOMICITY : all commands in a transaction succeed, or none do.

- implies that order of transactions doesn't matter
-

→ with RDBMS, because of the ACID property, we don't really have to worry ab the program crashing in the middle of when its running

* C: Consistency – after a commit, the database constraints must be satisfied (else transaction won't suceed)
    - for ex, size limits on int fields, fields that have to have each val. unique, etc.
* I: Isolation – concurrent transactions appear isolated from one another
    - aka it is synchronous ?? idk
* D: Durability – when a commit succeeds, its data is safely stored

SQL Alchemy

RECAP: What layers of the tech stack have we learned so far?

→ From top to bottom:

1. Angular Components – what the users see on each page
2. Angular Services – help to fetch & update data for your application
3. FastAPI – expose data to Angular services across HTTP
4. Backend service layer – called by the APIs to manipulate data

(1–2: frontend; 3–4: backend)

What is the last functionality still missing?

→ A place to store data such that it **persists** (is saved forever)

→ Currently, our data doesn't save – if we refresh our page/server or restart our project, all the data we worked with dissappears.

What component fulfills this need?

→ the **database**! A durable container that stores our data such that it stays intact, whether we

- → refresh the page
- → restart our project
- → or update our live deployment (on Cloud Apps)

→ Our storage for **persistent data**

→ the core component of the **final layer** of our tech stack.

What kind of database will we use?

→ a PostgresSQL relational database

What is a relational database?

→ a database that stores data in **tables with rows and columns**

- each column represents a field of data, and has a distinct, defined data type
- rows represent entries of data

What is a "primary key"?

→ a unique identifier/value that serves as a sort of ID for each row.

→ Each row usually has a primary key

Example of a table?

Table user :

PID(*)	name	ONYEN
111111111	Sally Student	sstudent
999999999	Rhonda Root	rroot

→ the fields are PID, name, and ONYEN

→ the (*) denotes that a field is a **primary key** & can be used to identify a row.

How do we interact with relational databases?

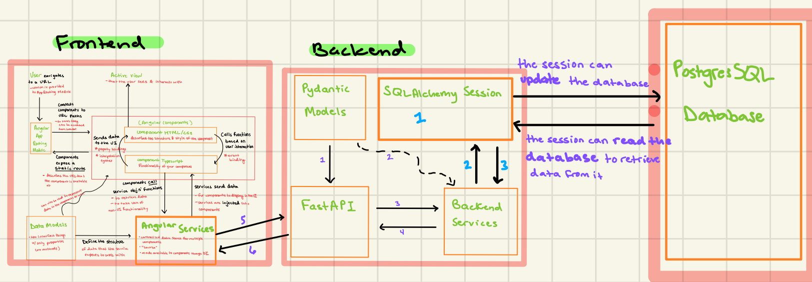
→ Using **SQL (Structured Query Language)**, a declarative language used for database manipulation and creation.

- we can use it to store & process info within a database.

How do we use SQL?	→ We can grab every entry from a table with the expression SELECT * FROM <table name> or grab the data of a specific entry by referring to its primary key: SELECT * FROM <table name> WHERE <primary key field name> = <primary key id> Ex SELECT * FROM user WHERE pid = 999999999
What is SQLAlchemy?	→ the primary "SQL toolkit" we'll use to interact with our (PostgresSQL) database. → A "Python Library" → allows us to connect to our SQL database from Python to manage data. • kind of a bridge between the existing (Python) backend services and the data in our database • allows us to perform CRUD (create, read, update, delete) operations → A much better option than writing & executing pure SQL queries in Python
Why is SQLAlchemy a better option than manually writing & running pure SQL queries?	→ Many reasons: → SQL queries would be translated as strings in Python, so we have to then manually build them up with concatenation & etc. • this is messy & error prone • VS SQLAlchemy: handles the SQL query creation process for us — we just have to call certain methods to perform desired actions. → running string SQL queries on the database is difficult & would require us to write a lot of extra service code. → Security risks: Malicious SQL queries could easily be run on the database • could cause major issues. • VS SQLAlchemy: mitigates such attacks & handles which data should & shouldn't be accessible → SQLAlchemy handles the conversion between SQL data and Python objects • aka puts the data in a "Python format", which makes it super easy to interact with & write to/modify the data (that we receive from the database) in Python. → with SQLAlchemy, we don't have to worry about what type of SQL database is in place - standardized way to interact w/ any of them. • there are diff types of SQL databases - like PostgresSQL, MySQL, SQLite, etc. — which have slight variations, either in features, the data types they support, and so on.

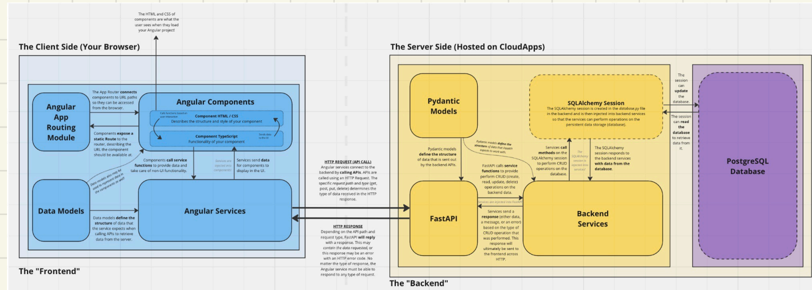
How does the addition of the PostgresSQL Database change our stack diagram?

→ 2 new additions: the database, and the SQLAlchemy session in the backend!



1. Description of the SQLAlchemy Session:
   - created in the database.py file in the backend
   - is then injected into backend services so that they can perform operations on the persistent data storage (database)

     (Ohh so this is where "backend services" actually gets the data from!!)
2. Backend Services call methods on the SQLAlchemy Session to perform CRUD operations on the database.
3. The SQLAlchemy Session responds to the backend services with data from the database



— SQLAlchemy Core and ORM —

→ SQLAlchemy is a Python Library whose functionality is broken up into 2 parts: the Core and the ORM.

What is the SQLAlchemy Core?

→ contains the base features that allow SQLAlchemy to function as a "database toolkit";
- the logic needed to run SQL queries on the database & to retrieve results — this means that we'll never have to write plain SQL code in our backend files.
- features to manage a constant connection with your database, using the SQLAlchemy Engine.

What is the SQLAlchemy ORM?

→ ORM = Object Relational Mapper

→ the ORM extends the base functionality of the Core to add OR mapping functionality.

	→ Enables you to easily transfer data between your SQL database and the Python backend by converting data from the **SQL Tables** (in SQL format) into the format of a traditional Python object called an **entity**.
What is the 'SQLAlchemy session'?	→ an object that you call to run SQL commands using Python objects • eliminates need for pure SQL code (to run commands) → Included in the ORM.

– SQLAlchemy Entities –

What are **entities**?	→ RECALL: the PostgresSQL database represents all of our data in the form of tables (columns = data fields and rows = data entries) → Entities are basically a way to represent the tables of the database in Python code as a Python object that we can then work with in the backend • the database is in a diff language; we can't just read from it using Python code/methods — we first need to convert it into a format that Python understands.
What do we use SQLAlchemy entities for?	→ to represent the expected shape of what our tables should look like — so that if we wanted to write/add to the SQL database (like creating new tables), we can use the Entity/structure as a guide. → to take data that we read from the database, & represent it as an object of this structure – aka an entity object.
How do "entities" actually look in Python?	→ In actuality, entity structures exist as Python classes → Each entity Python class serves to represent <u>one</u> table from the database. → Entity classes will need to use certain methods to map data from the database (and such tasks) – these are provided by the SQLAlchemy ORM! • specifically, we can import the DeclarativeBase class from the ORM • all of our entity classes should **extend from** (RECALL: Inheritance, subclassing, etc.) DeclarativeBase
How should we structure our entity classes within the backend?	1. create a class that serves to act as a superclass for all of our entities • the class inherits DeclarativeBase but otherwise empty – its just for structural purposes. entity_base.py:

```python
from sqlalchemy.orm import DeclarativeBase

class EntityBase (DeclarativeBase):
    pass
```

- importing from the SQLAlchemy ORM
- Python syntax to declare a class that a class is extending from (class Avocado extends IngImpl)

2. create entity classes for each table in the database. They should extend from EntityBase.

How do we create an entity class to make a table for the database?

→ When we create entities, we are mapping Python class fields to SQL relational database columns.

→ Ex creating an entity to represent the organization table in the csxl database

How we want the database table generated by OrganizationEntity to look:

Table organization

id (*) (int)	name (str)	description (str)	public (bool)
~	~	~	~
~	~	~	~
~	~	~	~

1. Map the entity class to a table from the database using the syntax

__tablename__ = "name of table"

2. Add a class field for each column that we want the table to have, using SQLAlchemy syntax.

field_name : Mapped [Python d.t. of field] = mapped_column (SQL d.t. of field)

- SQL RDBs have their own data types that correspond to Python ones:

Python		SQL
str	→	String
int	→	Integer
bool	→	Boolean

What is mapped_column()?

→ A function from SQLAlchemy ORM. In addition to data type, it also takes other optional arguments that specify details about each column, such as

- Boolean primary_key : denotes if a column is a primary key.
- Boolean nullable : defines whether a column's entries are allowed to be blank.
- Boolean autoincrement: allows a field's values to be automatically populated in increasing order
- ____ default : defines a default value to be filled in for every entry's value for that column. Type should be same as the field data type.

Example of an entity class for the "organization" table?

organization_entity.py

```
from sqlalchemy import Integer, String, Boolean
from sqlalchemy.orm import Mapped, mapped_column
from .entity_base import EntityBase


class OrganizationEntity(EntityBase):

    __tablename__ = "organization"

    id: Mapped[int] = mapped_column(Integer, primary_key
                                    = True, autoincrement=True)
    name: Mapped[str] = mapped_column(String, nullable=False,
                                      default = " ")
    description: Mapped[str] = mapped_column(String)
    public: Mapped[str] = mapped_column(Boolean, nullable=False,
                                        default = True)
```

- imports → required imports, including the superclass which inherits DeclarativeBase
- class line → entity class inherits the base entity superclass
- `__tablename__` → maps OrganizationEntity to a table in the PostgreSQL database named organization
- id, name, description, public → class fields

RECALL: What is a Pydantic Model?

→ Pydantic Models are data models that we use to define the structure of data that is sent to FastAPIs (like in response bodies)

- created as a Python **class object** that inherits Pydantic's BaseModel class and defines all the fields as well as the D.T.s they should accept (similar look to a TS interface)
- The Pydantic Model class is declared inside of the backend api.py file.
- for ex, a Pydantic Model for organizations:

```
class Organization(BaseModel):
    id: int
    name: str
    description: str
    public: bool
```

What is the difference between an Entity and a Pydantic Model?

→ Conceptually, they seem very similar - both serve to define the structure of and "represent" our application's data.

→ However, it is important to separate these 2 items because there are cases where the data that the API exposes & what data is acc stored in the database are different from one another.

What is the difference between an Entity and a Pydantic Model?

Pydantic Models

→ represent the shape of data transferred **by the backend's API**;

- data sent to the API from the client/frontend via request bodies.
- data sent to the API from the backend service via Service Function calls.

→ exist as class objects that inherit from BaseModel — aka "API-Formatted"

SQLAlchemy Entities

→ represent the shape of data used **by the PostgresSQL database**

- data sent to the database (creating new tables; updating tables)
- data retrieved/read from the database

→ exist as class objects that inherit from DeclarativeBase — aka "SQL Alchemy-formatted"

So how do we retrieve data from the PostgresSQL Database to access in the API?

→ FastAPI only works w/ Pydantic Models -- so we can't just pass over the SQLAlchemy Entities that were created

→ Instead, the Backend Service first calls methods on the SQLAlchemy Session to receive data in the form of **entities**

- Then, it **converts** the entity into a corresponding **Pydantic Model**
- Finally, it sends this Pydantic Model to FastAPI when requested.

How do we add data from the API to the PostgresSQL Database?

→ **vice versa**: when we need to **add** data received from the frontend (via FastAPI) to the SQL database (via SQLAlchemy Session):

- the Backend Service converts Pydantic Models that it receives from FastAPI into entities
- then, it sends these entities to the SQLAlchemy Session by calling methods on it.

How does the Backend Service execute these conversions?

→ By calling helper functions that we can define inside of our SQLAlchemy Entity class (aka OrganizationEntity)!

Method to convert an entity into a model?

→ To convert an existing instance of an entity into a new Model object:

- create an instance function in the entity class

organization_entity.py

```
class OrganizationEntity (EntityBase)
  //... (other code)

  def to_model (self*) -> Organization:
      return Organization (
          id=self.id, name=self.name,
          description=self.description,
          public=self.public)
```

- specifies the return type
- creates & returns a new Organization (which is the data model) object where all of the fields are set to the values of the corresponding fields in the current instance of the Org Entity object.

What does the self keyword mean in this Python class?

→ Refers to the current instance of the class — kind of like the "this" keyword in Java.

→ the self keyword as a parameter indicates that the function will be using the current instance of its class object (aka an OrganizationEntity object) inside the method body.

→ it's sort of an "invisible" parameter — when calling a function that specifies self as a param, we don't actually have to pass anything in

- b/c we are already calling the function from an existing instance of the class; no new info needs to be passed in
- i guess its more of some syntactical/annotative thing specific to Python? idk
- In Java, we never ask for "this" as a param. b/c its redundant ... (?)

How would we use to_model ?

→ In the backend service, say we have an existing OrganizationEntity object "entity1":

```
my_model : Organization = entity1.to_model()
```

method to convert a model into an entity?

→ To convert an existing Pydantic Model into a new entity object, we can define a **static method** in the Entity class that takes the Model object as a parameter:

organization_entity.py

```
class OrganizationEntity (EntityBase):
    //... other code

    def to_model (self) → Organization :
        (implementation hidden) //

    @classmethod
    def from_model (cls, model: Organization) → Self:
        return cls(
            id = model.id,
            name = model.name,
            description = model.description
            public = model.public )
```

in Python, static methods are declared with the @classmethod decorator

Self: indicates that the return type is an instance/object of the class (OrgEntity)!

What is the cls keyword in Python?

→ it refers to the class that a static method (@classmethod) is acting on.

→ Is automatically passed in as the 1st parameter for static methods.

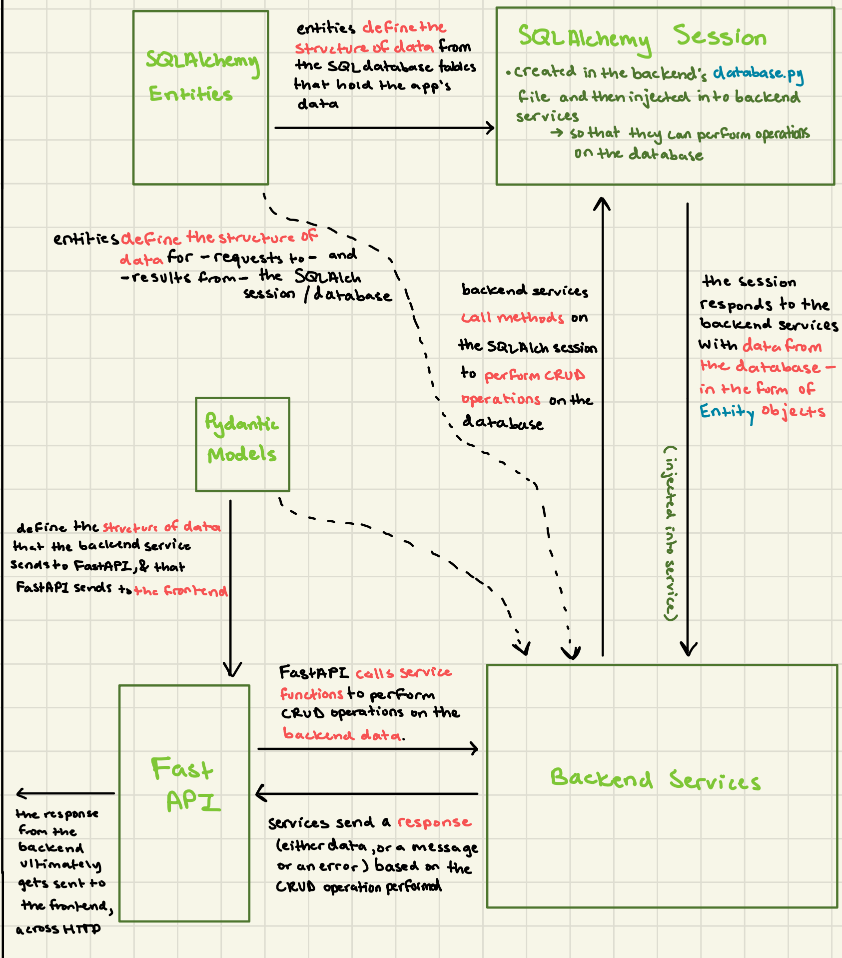
How would we use from_model ?

→ In the backend service class, say we have an existing Organization Pydantic model named "model1":

```
my_entity : OrganizationEntity = OrganizationEntity.from_model(model1)
```

model1: takes the model obj as a parameter

How do the SQLAlchemy entities & session fit into our backend stack?



— Connecting to the Database —

→ RECALL: When it comes to accessing the SQL database for our application, the **backend service** is the component that is ultimately responsible for making that connection — which it does by using **SQLAlchemy**.

How does the backend service access SQLAlchemy?

→ By using a **SQLAlchemy Session**: the object that actually establishes conversations with the database — this is **the object that we use to interact w/ our database from the backend service**

→ We expose the SQLAlchemy Session as an <u>object</u> that can be **injected** into each service.

Where is the "session" created?

→ In a separate backend file (for csxl, its called database.py), where an engine object is created to establish the initial connection with our database.

What is the SQLAlchemy Engine?

→ A feature of the SQLAlchemy Core that enables us to maintain a stable connection to the database

- It manages a constant connection (with the DB)

→ The engine reads our application's .env environment files which contain important, secret(?) information about how to access the database

→ For ex, here is a code snippet from database.py where the engine is established:

```
def _engine_str(database = getenv("POSTGRES DATABASE")) → str:
    ...
        user = getenv ("POSTGRES_USER")
        password = getenv ("POSTGRES_PASSWORD")
        host = ...
        port = ...
        return "..."

engine1 = sqlalchemy.create_engine(_engine_str(), echo=True)
```

How is the SQLAlchemy Session created in database.py?

→ with a function, db_session, that uses the engine object to return a Session object. It creates a singular, shared instance of the session, for all services to use.

- In database.py, right under the above code snippet,

```
def db_session():
    session = Session(engine)
    try: yield session
    finally: session.close()
```

How do we inject the Session into our backend service?

→ By using the Depends() syntax to inject it into the backend service's initializer method:

```
class OrganizationService:
    def __init__(self, session: Session = Depends(db_session)):
        self._session = session
    ...
```

initializes the 'OrganizationService' session

→ We can now access the shared db_session using the service class' ._session field!

CRUD Operations on the SQL Database

What are the CRUD operations in the context of SQL database?

→ Create : Add new rows to the table
Read : Retrieve existing rows from the table
Update : Modify existing rows on the table
Delete : Remove rows from the table

→ RECALL : CRUD for request types that the frontend sends to the backend API:

Create : POST
Read : GET
Update : PUT
Delete : DELETE

→ We perform CRUD operations in the backend service file(s), using the SQLAlchemy session in our entity.

What is a transaction?

→ What SQLAlchemy uses to perform CRUD operations

→ Purpose / Main idea: to denote an all-or-nothing collection of changes to the database, meaning that either:

- all of the requested changes should happen to the database, or
- none of the changes are performed — e.g., if something happens to cause any of the changes to fail.

→ Super important because it ensures that the database is always in a consistent state even if errors occur (like a power outage, connection dropped in middle of transaction, failure of a modification, etc.)

— Reading Data —

How do we read data from the database?

→ By creating a query : a request for data.

→ We create queries using the select function imported from sqlalchemy, and we pass in the entity class that represents the desired table

- bc at this point, we have already created entity classes (like OrganizationEntity) for every table in the PostgresSQL database.

How do we retrieve all of the data from a particular table in the DB?

→ EX in the backend service class "OrganizationService" (in /backend/services/organization.py)

1. create a query for the desired table.

```
from sqlalchemy import select
query = select(OrganizationEntity)
```

→ the required import

2. Use the service's session object to find all of the rows:

```
entities = self._session.scalars(query).all()
```

- self._session: RECALL that we injected the SQLAlchemy session into OrgService's initializer, and refer to it using this field.
- .scalars(query): rows are denoted as "scalars"
- .all(): use .all() to return all of the scalars (rows) that match the query

What is the data type returned by self._session.scalars(query).all()?

→ A list of entity objects!

→ RECALL: a SQLAlchemy Entity class represents a table in the database, while every instantiated Entity object represents an entry in the SQL table that the class defines.

3. Convert the list of **entity** objects into a list of pydantic **model** objects using the already defined `to_model()` function

(and then put all of this into a method in the backend service!):

'Final' version of OrganizationService:

```
class OrganizationService:

    def __init__(self, session: Session = Depends(db_session)):
        self._session = session


    def to_model(self) -> Organization:
        ...

    @classmethod
    def from_model(cls, model: Organization) -> Self:
        ...

    def allData(self) -> list[Organization]:
        query = select(OrganizationEntity)
        entities = self._session.scalars(query).all()
        models = [entity.to_model() for entity in entities]
        return models
```

returns a list of Organization Pydantic Model objects

Python syntax trick that performs the following for loop in one line:

```
models = []
for entity in entities:
    models.append(entity.to_model())
```

How do we retrieve all of the data that **matches some condition**?

→ Using the **query builder**: instead of passing the table's entity class into `select()`, we pass it into the session object's `session.query()` function

→ Then, we add **filter** conditions to our query using `session.query(__).where()`

→ Finally, after adding filters, call `.all()` to get all rows of data that match the filter.

Example?

→ Say we wanted to retrieve all rows of the organization table where "public" = True:

in OrganizationService

```
def all_public(self) -> list[Organizations]:
    entities = self._session.query(OrganizationEntity)
                .where(OrganizationEntity.public == true)
                .all()
    return [entity.to_model() for entity in entities]
```

How do we retrieve a single entry from a table?

→ By querying the element based on its **primary key**, using the SQLAlchemy Session's `.get(x, y)` method, where

- x = the desired table's entity class
- y = the primary key

Example?

→ Say we want a function to receive an organization table entry **by its ID**:

```
def get_by_id(self, id: int) -> Organization:
    entity1 = self._session.get(OrganizationEntity, id)
    return entity1.to_model()
```

- takes in the desired ID number as a parameter
- `.get()` returns a **single entity**, so we return a single Organization model

— Writing Data —

How do we write (add) new data to the database?

→ Using the SQLAlchemy Session's `.add()` function!

→ [Ex] Say we want to add a new organization to the "organization" table in the database.

1. In the **backend service**, create a new `.create()` function that takes in a new organization as a parameter.
   - Since we are receiving the new data to be added from the frontend via the API, this parameter will be in the form (D.T.) of a **Pydantic Model**:
2. Convert the model input into an **entity** and use `.add()` to add it to our **transaction**:

```
def create(self, org: Organization) -> Organization:
    entity1 = OrganizationEntity.from_model(org)
    self._session.add(entity1)
```

converting model to entity using the OrgEntity class' **static** conversion method.

Has the data successfully been added to the DB at this point?

→ **No**, not yet. Because `.add()` mutates the state of the database, it first appends the action to the current transaction, to be committed with any other changes in accordance w/ the transaction **all-or-nothing** principle.

→ Similar to staging and then committing in Git.

How do we execute the "transaction"?

3. Call the Session's `.commit()` method (in the `create()` function):

(code Ex on next page)

"Final" version of OrganizationService:

```
class OrganizationService:

    def __init__(self, session: Session = Depends(db_session)):
            self.__session = session

    def to_model(self) -> Organization:
            ...

    @classmethod
    def from_model(cls, model: Organization) -> Self:
            ...

    def allData(self) -> list[Organization]:
            ...

    def create(self, org: Organization) -> Organization:
        entity1 = OrganizationEntity.from_model(org)
        self.__session.add(entity1)
        self.__session.commit()
        return entity.to_model()
```

Now, the database is updated (→ self._session.commit())

we have our function return the object that we created — in model form — to ensure that it has been created correctly

— Deleting Data —

How do we delete data from the database?

→ Using the SQLAlchemy Session's .delete() function!

→ .delete() takes in the table entry — aka a row; a single entity object — that should be deleted from the database.

- Similar to .create(), we must commit the delete action to execute the transaction.

Example?

→ Ex Say we want to delete an organization from the "organization" table in the database, by its ID.

1. Obtain the entry / object that you want to delete using the session's .get() function (RECALL from "reading data") section.
2. Pass that entity into .delete()
3. Call the session's .commit() to execute the action.

(backend service) OrganizationService:

```
class OrganizationService:
    ...

    def delete_by_id(self, idNum: int):
        entity1 = self._session.get(OrganizationEntity, idNum)
        self._session.delete(entity1)
        self._session.commit()
```

—Advanced Querying Techniques (for reading data)—

How can we make a query for data that matches multiple conditions?

→ RECALL that we used `.query(Entity Class).where(condition)` to retrieve all of the data from a table that matched one condition.

→ To create a query for multiple conditions which all have to be true — basically the boolean && logic — we can either:

A. use multiple `.where()` calls one after the other in the same line, or

B. Pass all of the conditions into a single `.where()` call, as multiple arguments.

Example?

→ Retrieving all organizations that are public & that have "Carolina" in the string name in the 'name' field:

A.

```
entities = self._session.query(OrganizationEntity)
    .where(OrganizationEntity.public == true)
    .where("Carolina" in OrganizationEntity.name)
    .all()
```

- OrganizationEntity → the table that we want to search
- in → the "in" keyword is used to search string-type entries for a keyword

B.

```
entities = self._session.query(OrganizationEntity)
    .where(OrganizationEntity.public == true, "Carolina" in OrganizationEntity.name)
    .all()
```

How can we make a query for data that matches either of multiple conditions?

→ A.k.a., boolean "OR" logic (like the || operator)

→ To create a query for all data entries where at least 1 of multiple specified conditions, we can use the Python OR operator, "|":

EX Query for all organizations that are either public OR have "Carolina" in their name (or both).

```
entities = self._session.query(OrganizationEntity)
    .where((OrganizationEntity.public == true) | ("Carolina" in OrganizationEntity.name))
    .all()
```

→ When using the "OR" (|) operator, all conditions must be surrounded by parentheses or unexpected errors could occur.

What if the conditions are alternative values for the same field?

→ We can either do the same syntax as above, OR use the `.in_()` method shorthand.

EX query for organizations with either of 2 org names:

```
entities = self._session.query(OrganizationEntity)
    .where(OrganizationEntity.name.in_(["CS+SG", "VR Club"]))
    .all()
```

→ pass the options into `.in_()` as arguments

Database Relationships

What are database relationships?

→ They define the connections between the tables in a database, and allow for tables to reference other tables (like pointing to other tables' entries, for ex)

→ 3 main types of database relationships: **one-to-one**, **one-to-many**, and **many-to-many** relationships.

→ Depending on the relationship you want to establish, you'll need to modify your SQLAlchemy Entity classes for the respective tables accordingly.

What is a one-to-one D.B. relationship?

→ Each item in one table points to **at most one item** in another table, and vice versa.

→ EX Organizations and their Presidents.

- concerns the "organizations" and "user" tables
- Each organization only has 1 president, and each user can be the president of at most 1 organization.

What is a one-to-many D.B. relationship?

→ Each item in one table points to **many** items in another table, but items in the other table can point to **at most one** item in the original table

→ EX Organizations and the Events that they host.

- concerns the "organizations" and "events" tables.
- Each organization can host numerous events, but each event only has 1 organization that hosts it.

What is a many-to-many D.B. relationship?

→ Each item in one table can point to many items in another table, and vice versa

→ EX Events and their registered attendees (users).

- concerns the "events" and "users" tables
- Each event can have many registered users, and each user can also register for many events at once.

— Implementing a One-to-One Relationship —

→ Lets use the "organizations & presidents" example from above. We can imagine the 2 tables in the SQL Database to look something like this:

Table organization

id (*) int	name str	description str	public bool	(other various fields)...

Table users

pid (*) int	name str	(other various fields)...

→ RECALL that each of these tables is represented by an OrganizationEntity and UserEntity class (respectively), where each column in the table is a **Field** in the entity.

→ RECALL that "*" denotes a primary key — the unique identifier for a row in the table.

What is a **foreign key**?	→ In a database, it is a field (column) that **refers to the primary key of another table**. • thus **allowing you to reference records in a different table based on their unique ID** → Creates a relationship between 2 tables in a DB → The building block for database relationships!
Where do we implement the idea of a "foreign key"?	→ If we want one DB table's entries to reference another's, then we would want each entry to store the primary key value of the other table's entry that it is "linked to" • This is the purpose of a **foreign key column / field**. → Ex: Adding a field to the organization table that stores the PID (which is the PK* of the user table) of each organization's president:

Table organization

id (*) int	name str	public bool	(various fields) ..			president_pid (←) int
~	~	~	-	-	-	~
~	~	~	-	-	-	~
~	~	~	-	-	-	~

→ The ← symbol denotes a foreign key field.

So how do you add a foreign key field to a table's Entity?	→ RECALL that we use mapped_column() to define fields in our table: `id: Mapped[int] = mapped_column(Integer, primary_key=True, autoincrement=True)` → To generate a column that contains a foreign key, we can pass a "ForeignKey()" object into the column via mapped_column • ForeignKey( ) takes the column of the other table you want to reference as its parameter, in the format "table.field"
Ex?	→ In organization_entity.py:

```
class OrganizationEntity(EntityBase):
    __tablename__ = "organization"
    id: Mapped[int] = mapped_column(Integer, primary_key
                                    = True, autoincrement=True)
    name: Mapped[str] = mapped_column(String, nullable=False)
    president_pid: Mapped[int] = mapped_column(ForeignKey("user.pid"))
```

passes in the "pid" column of the "user" table

How do we actually access the entity object from another table?	→ Just creating a foreign key field isn't enough — each organization entry needs to be able to access the UserEntity object that its "president_pid" field refers to → **We establish this connection using SQLAlchemy relationship fields.**
What is a **relationship field**?	→ Fields in the **entity class** that **do not exist in the table as columns** — but their values are automatically populated by SQLAlchemy when data is being read. → This is how we populate our entities with data from relationships.

What type of object is a relationship field?

→ relationship fields are, just like any other field, basically a list of data – it just isn't displayed in the D.B..

→ For ex, OrganizationEntity.name is an object which is a list of all the names in the "name" column of the table. Similarly, OrganizationEntity.president_pid is a list of all the organization entries' respective president pids.

How do we define relationship fields that point to one another?

→ In each entity class, we create a relationship field that points to the name of the other entity's relationship field

→ To create a relationship field, **we use the relationship() method rather than mapped_column().**

→ **Syntax**: <field name> : Mapped ["other Entity class' name"] = relationship (back_populates = "other relationship field's name")

Ex?

→ Here are our finalized OrganizationEntity and UserEntity classes after establishing the one-to-one relationship:

OrganizationEntity

```
class OrgEntity (DeclarativeBase):
    __tablename__ = "organization"                                   ] tablename

    id : Mapped [int] = mapped_column (Integer, primary_key           ]
                                       = True, autoincrement=True )   ]
    name: Mapped [str] = mapped_column (String, nullable = False )    ] fields
    president_pid : Mapped [int] = mapped_column (ForeignKey ("user.pid"))  ]

                                                                      relationship field
    president: Mapped ["UserEntity"] = relationship (back_populates = "president_for")
```

Stores the user data of an organization's president. (→ president field)

Mapped ["UserEntity"]: says that the data type of the president field is a UserEntity object.

"president_for": pointing to the relationship field in UserEntity.

→ SQLAlchemy is smart enough to know to populate the president field with the UserEntity object with the same PID as the value stored in the president_pid field.

UserEntity

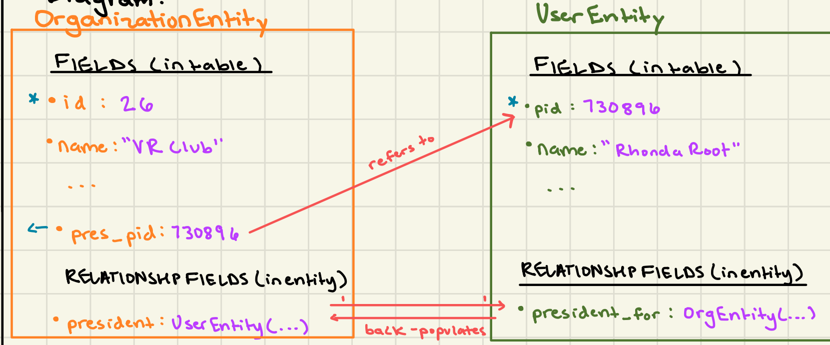
```
class UserEntity (DeclarativeBase):
    __tablename__ = "user"

    pid : Mapped [int] = mapped_column (Integer, primary_key
                                        = True, autoincrement=True )
    name: Mapped [str] = mapped_column (String, nullable = False )

    president_for : Mapped ["OrganizationEntity"] = relationship (back_populates = "president")
```

Stores the organization data for an organization that a user is the president of. (→ president_for field)

"president": pointing to the relationship field in OrganizationEntity

Summary of one-to-one relationships?

→ Diagram:



→ Steps:

1. Define/create a foreign key field in one table that refers to the PK field of the other table.
2. Create relationship fields in both tables who point to each others' names.

— Implementing a One-to-Many Relationship —

→ Example: Organizations, which can host multiple events, and Events, which can only be hosted by <u>one</u> organization each.

→ We can imagine the organizations and events tables in the DB to look like this:

Table organization

id (*) int	name str	description str	public bool	(other various fields)...				
~	~	~	~	~	–	–	–	~
~	~	~	~	~	–	–	–	~
~	~	~	~	–	–	–	–	–

Table events

id (*) int	name str	(other various fields)...					host_org_id (←) int
~	~	~	–	–	–	–	~
~	~	–	–	–	–	–	~
~	~	–	~	–	–	–	~

How do we implement a one-to-many relationship?

→ Almost identical to setting up a 1-to-1 relationship, **except** the entity on the "one" side stores a **list** of the other entity's objects, rather than a single entry.

→ STEPS:

1. Define the table/entity that represents the "one" side & the "many" side, respectively.

- EX "one" side : OrganizationEntity
  "many" side : EventEntity

(Since <u>1</u> organization can have <u>many</u> events associated with it)

2. Add a **foreign key** column/field to the "many"-side entity.

- EX such that, for each event entry, the field contains the PK of the singular corresponding organization that hosts it.
- in `event_entity.py` :

```
host_org_id : Mapped[int] = mapped_column(ForeignKey("organization.id"))
```

- this establishes a one-to-one relationship between the <u>event</u> & org tables.

3. Create a **relationship field** in the "many"-side entity that stores a **single instance** of the other table's entity class (aka a single table entry):

- in `event_entity.py`:

```
org : Mapped["OrgEntity"] = relationship(back_populates="event")
```

- For each event, this stores the hosting organization — populated automatically by SQL Alchemy using the foreign key column we defined in step 2.

4. Create a **relationship field** in the "one"-side entity that stores a **list of instances** of the other table's entity class (aka multiple entries that all correspond to one entry in the "one"-side entity / table).

- it should point to the name of the relationship field made in step 3 (via "back_populates=")
- in `organization_entity.py`:

```
event: Mapped[list["EventEntity"]] = relationship(back_populates="org")
```

- For each organization, this stores the list of events that it hosts.

Why do we only add a foreign key column to one entity (table)?

→ In a one-to-many relationship, we only add a foreign key column to the entity on the "many" side (e.g. the entity who can store at most one connected entry from the other table, like Event Entity)

→ This is because Postgres SQL has no functionality to store a list of foreign keys as a field — it can only store one item (per field per entry).

- And if we wanted to put a foreign key column in the OrganizationEntity table, for example, then we would need to store a list of IDs in it.

What would our final EventEntity and OrganizationEntity classes look like?

EventEntity

```
class EventEntity(DeclarativeBase):
    __tablename__ = "event"

    id : Mapped[int] = mapped_column(Integer, primary_key
                                     = True, autoincrement=True)     ] normal
    name: Mapped[str] = mapped_column(String, nullable=False)       ] fields
    host_org_id: Mapped[int] = mapped_column(ForeignKey("organization.id"))]  foreign key field

    org : Mapped["OrgEntity"] = relationship(back_populates="events")]  relationship field
```

OrganizationEntity

```python
class OrgEntity (DeclarativeBase):
    __tablename__ = "organization"

    id: Mapped[int] = mapped_column(Integer, primary_key
                                    = True, autoincrement = True)
    name: Mapped[str] = mapped_column(String, nullable = False)

    events: Mapped[list["EventEntity"]] = relationship(back_populates="org")
```

relationship field

—Implementing a Many-to-Many Relationship—

→ Example: Events and their registered attendees (users)

Why is implementing a many-to-many relationship more complicated?

→ RECALL that PostgresSQL cannot store lists of foreign keys as a field—just single items.

- In the previous relationships, we had every organization map to a single user (via its foreign key). And then we had every event map to a single organization (even though there were multiple events which mapped to the same foreign key value)
- So this didn't matter.

→ But now, we need **both tables** to be able to refer to several items in one entry (e.g., a single user should be able to store a list of events and vice versa), so there is no adequate place to put the foreign key field

- Therefore, we **cannot** directly establish a many-to-many relationship between 2 entities on their own.

So how do we connect the 2 entities?

→ By creating an **association table** to provide a method to connect items from one table to items in another.

What is an Association table?

→ A table that we **add to our SQLDatabase**, that matches together the IDs from 2 different tables.

→ Each record/entry in an A.T. serves to store an explicit relationship between 2 records.

What might the association table look like in the database?

→ For this example with users & events, we can imagine something like this:

Association Table

id(*)	event id (←)	user id (←)
1	1	1
2	1	2
3	2	1
4	3	2
5	3	3
6	4	5

→ This table demonstrates a **many-to-many** relationship; notice how event id=1 maps to 2 items in the user entity, while user id=2 maps to 2 items in the event entity.

→ Also, the association table now becomes the place where we store our **foreign key columns**.

How do we create an association table?

→ By making a new **entity** class for it!

- For the example, we can create an EventRegistrationEntity class.

What are the steps to implementing a many-to-many relationship via an association table?

1. Create your assoc. table's entity, adding **foreign key fields** to store each entities primary keys.

```
class EventRegEntity(EntityBase):
    __tablename__ = "event-registration"

    id: Mapped[int] = mapped_column(Integer,
                          primary_key=True, autoincrement=True)
    event_id: Mapped[int] = mapped_column(ForeignKey("event.id"),
                          primary_key=True)
    user_pid: Mapped[int] = mapped_column(ForeignKey("user.pid"),
                          primary_key=True)
```

our main PK field

- notice that we make the 2 foreign key fields into **primary keys** as well, since these 2 fields together uniquely identify each registration.

2. Create **relationship fields** in all 3 classes such that:
   - the 2 o.g. tables/entities each have a relationship field storing a **list** of the A.T. entries/objects
     - each field should back_populate their respective relati. field in the A.T. entity.
   - the A.T. entity has **2 relationship fields**, one to back populate each of the 2 entities.

→ In EventEntity:

```
registrations1: Mapped[list["EventRegEntity"]] = relationship(back_populates="event",
                                  cascade="all, delete")
```

→ In UserEntity:

```
registrations1: Mapped[list["EventRegEntity"]] = relationship(back_populates="user",
                                  cascade="all, delete")
```

→ In EventRegEntity (the A.T. class):

```
event: Mapped["EventEntity"] = relationship(back_populates="registrations1")
user: Mapped["UserEntity"] = relationship(back_populates="registrations2")
```

What is "cascade = "all, delete" "?

→ A rule that ensures that when you delete an event or user entity object, all of the registrations to it in EventRegEntity are also deleted.

- all rows which contain the deleted object's ID will be deleted.

→ Important because it prevents fractured relationships when entities are deleted.

3.

Authentication and Authorization

What is authentication?

→ verifying **who** the "subject" accessing a system is.

- A subject can be a person, or an automated program accessing a system on behalf of a person, group, or organization. In the context of the csxl website, we take "subject" to be a person using the CSXL application.

→ Authentication in the csxl.unc.edu website is done with the UNC SSO (single sign-on) service.

What is authorization?

→ Verifying **what** a certain subject has permission to do

→ i.e., verifying whether a subject/user **has permission** to carry out an action on a resource within the system.

→ For Ex, the leader of a workshop may have permission to edit a workshop's details, whereas a registered participant/attendee may not.

How is authorization managed within the csxl website?

→ By following the union of 2 distinct rule sets:

1. Feature-specific rules
2. Administrative Permission rules

What are Feature-specific rules?

→ For any feature of a website which is related to one or more users in the system (e.g. via one or more of its T.S. "models"), these users usually need some authorization to carry out specific actions on those Models.

→ Feature-specific rules are how we achieve that authorization - guidelines ('rules') that specify which actions can be carried out by which users.

Where do we enforce the logic of feature-specific authorization rules?

→ Where: methods in the feature's **backend service** (.py) file!

→ RECALL: The purpose of the backend service is to define methods like `get_timer(self, subject: User, timerID: int) -> list[PomodoroTimer]`, which:

ⓐ in their implementation, call methods from the **SQL database** to retrieve data (aka call methods defined in **entity** (Python) classes, for ex `entity1 = self._session.get(PomodoroTimerEntity, timer_id)`)

ⓑ Get called by the **FastAPI** (python) class, to which they return data from the SQL database after converting it from entity to model format.

How do we enforce authorization in the backend service?

→ **to start**: All backend service layer methods where authorization is a concern should accept a user as their first parameter, in the format:

```
someMethod(subject: User) { }
```

→ This parameter represents the user attempting to carry out the action (and whose authorization needs to be verified).

What exception should the backend service method raise?

→ Additionally, if our backend service layer determines that the subject passed in (by the API calling said backend service method) does <u>not</u> have permission to carry out the operation, we should have the backend service **raise an exception.**

→ A "User Permission Exception", like this:

```
raise UserPermissionException('workshops.update', f'workshops/{workshop.id}')
```

1 — 'workshops.update'; 2 — 'workshops/{workshop.id}'

1. <service>.<method>, for the specific backend service's method where the error occurs
2. The <u>target path</u> of the primary model being operated on (minus the leading "/api" part).
   - In ex above, the FastAPI path to the model being operated on was "/api/workshops/1"

Quiz 2 Review

- SQLAlchemy readings (all chapters)
- Kris' 1st lecture on SQLAlchemy
- database relationships!!!!
- slides from canvas announcement
- "system design" practice qs
- gradescope questions

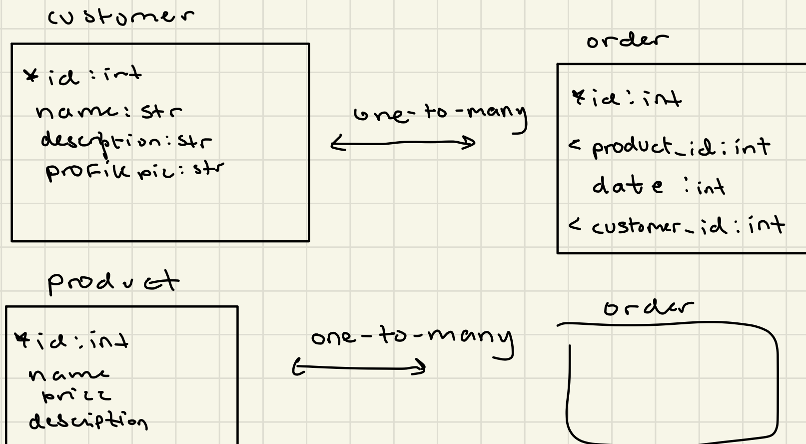
- RDBMS (Relational Database Management System):
  Databases / DB concepts, 'keys', transactions, ACID properties, etc.
- SQLAlchemy: ORMs, Session, Entities (vs Pydantic), relationships, etc

STILL

- ~~Steps to connecting everything~~
- ~~session/CRUD operations~~
- ~~gradescope qs~~
- Sys. Design q

  - customer ⟷ order
  - product ⟷ order
    one-to-many

- every order is of one prod. type, but one prod may have many orders made on it



RDBMS

→ Motivation: w/o database, our data (on our site) doesn't save when you refresh or restart the server.

a RDBMS (Relational Database Management System) provides **persistent storage** and is "where data lives."

→ RDBMS is "postgres", meaning separated from frontend/backend components.

→ What type of DB do we use? **Postgres SQL relational database**:

- "relational" → data stored in tables, where columns ≈ fields w/ defined data types and rows ≈ entries of data
- We interact with the database using SQL, a 'declarative' language used to make queries.

→ **ACID Properties** (RDBMS Transactions have them):

- **A**tomicity: Either a transaction successfully completes all of its commands, or none of them.
  - this property is why we don't have to worry about program crashing in the middle of a transaction causing unwanted changes to data - either all transactions executed, or none.
- **C**onsistency: After a commit, the database constraints must be satisfied (e.g. size limits on `int` fields, required fields, fields where each entry's value must be unique, etc.), OR the transaction won't succeed. (Ensures that data integrity constraints are not violated before & after transactions occur)
- **I**solation: Ensures that multiple transactions, even if concurrent, occur **independently** without interference.
- **D**urability: Ensures that data persists even if the DB system goes offline (When a commit succeeds, its data is safely stored)

SQLAlchemy

→ SQLAlchemy is an ORM (object relational mapper)

→ Our toolkit for interacting with the DB; A "Python Library" with 2 parts: **Core** and **ORM**

→ Allows us to interact w/ and perform CRUD operations on the DB in Python rather than SQL-lang queries (b/c it does the conversion for us).

→ **SQLAlchemy Core**:

- base features that allow SQLAlch to function as a "database toolkit" (like logic for running SQL queries)
- manages the constant connection to the database using the SQLAlchemy Engine

→ **SQLAlchemy ORM** (extends upon base functionality of the Core):

- functionality for object-relational mapping ("from sqlalchemy.orm import Mapped, mapped_column, relationship")
- functionality to convert data from SQL format to a Python Entity object
  - "from sqlalchemy.orm import DeclarativeBase" the class provided by SQLAlchemy ORM which contains methods for mapping data. All Entity classes should extend from it.
- The SQLAlchemy **Session** ("from sqlalchemy.orm import Session")

* Basically, everything comes from the ORM except the engine

Entities

→ DEFN: represent DB tables as Python code objects that we can then work w/ in the backend

→ A class represents a **table**; an instance represents an **entry**

→ Entity classes (e.g. event_entity.py) are in the **backend**

→ Ex:

```
class EventEntity(EntityBase):
    __tablename__ = "event"
    id: Mapped[int] = mapped_column(Integer, primary_key=True, autoincrement=True)
    name: Mapped[string] = mapped_column(String, default="")
```

- `__tablename__ = "event"` → how to declare name of a table
- `mapped_column` ↳ function to specify aspects of a column/field, including the d.t.

→ **primary key**: column w/ an int unique to each entry

→ **Foreign key**: field that refers to another table's primary key; building block for creating DB relationships

Database Relationships

- **One-to-One**:

  → e.g., organizations & their presidents (where a User can be pres. of max 1 organization)

  → Steps:

  1. Add foreign key field to just 1 of the 2 entity classes:

     ```
     pres_pid: Mapped[int] = mapped_column(ForeignKey("user.pid"))
     ```

  2. Add **relationship fields** to both entities:

     ```
     president: Mapped["UserEntity"] = relationship(back_populates="pres_for")
     ```

     and

     ```
     pres_for: Mapped["OrgEntity"] = relationship(back_populates="president")
     ```

- **One-to-Many**:

  → e.g., organizations & the many events they host

  → STEPS:

  1. Add a foreign key field to only the "many side" entity (e.g. EventEntity)
  2. Add rel. field to "many side" entity which stores a single instance of the other entity class (e.g. the singular org hosting each event)
  3. Add rel. field to "one side" entity storing a list of instances of the other entity

- **Many-to-Many**:

SQLAlchemy Session

→ The backend service connects to the DB using a SQLAlchemy Session object, which gets injected into each backend service class through their "initializer method"s, using Depends() function:

```
class OrgService:
    def _init_(self, session: Session = Depends(db_session)):
        self._session = session
```

→ The session provides all the methods to perform CRUD operations; backend service calls session's methods to do this, for ex:

```
self._session.scalars(query).all()
self._session.add(entity1)
self._session.delete(entity1)
self._session.commit()
```

self._session.commit() → mandatory to include, else "C" (create), "U" (update), or "D" (delete) operations won't actually get performed

Connecting DB to rest of Stack

1. Frontend service method makes HTTP Request to FastAPI to perform CRUD on data, expecting a TS Interface Model object
2. FastAPI calls the backend service's methods to perform the relevant CRUD operation, expecting a Pydantic Model object in return.
3. Backend services are the ones doing task of communicating w/ the DB. They call methods on the session in order to perform operations
   - C (e.g. FastAPI has sent new data to the backend method):
     1. The backend method takes the Pydantic Model it has received & calls the corresponding Entity Class' static "from_model" method which returns the same data converted into an entity object

        `entity1 = OrgEntity.from_model(model: Organization)`
     2. It then calls the Session's methods to add & commit the new entry to the table
     3. Returns the same model obj. back to FastAPI using to_model (verification purposes)
   - R (e.g. FastAPI is requesting some data from the backend):
     1. Backend serv. makes the query & receives desired data from the Session - as Entity object(s)
     2. It then calls the Entity class' to_model which converts the entity w/ the desired data into a Pydantic Model. It returns this PM obj. to the FASTAPI
4. FastAPI returns the requested data to the frontend service as a Pydantic Model.

artist —— song
one-to-many

user —— playlist
one-to-many

playlist —— song
many-many

user — artist
many many (Followers)