T2A1 - Workbook

Nick Ducker

Q1

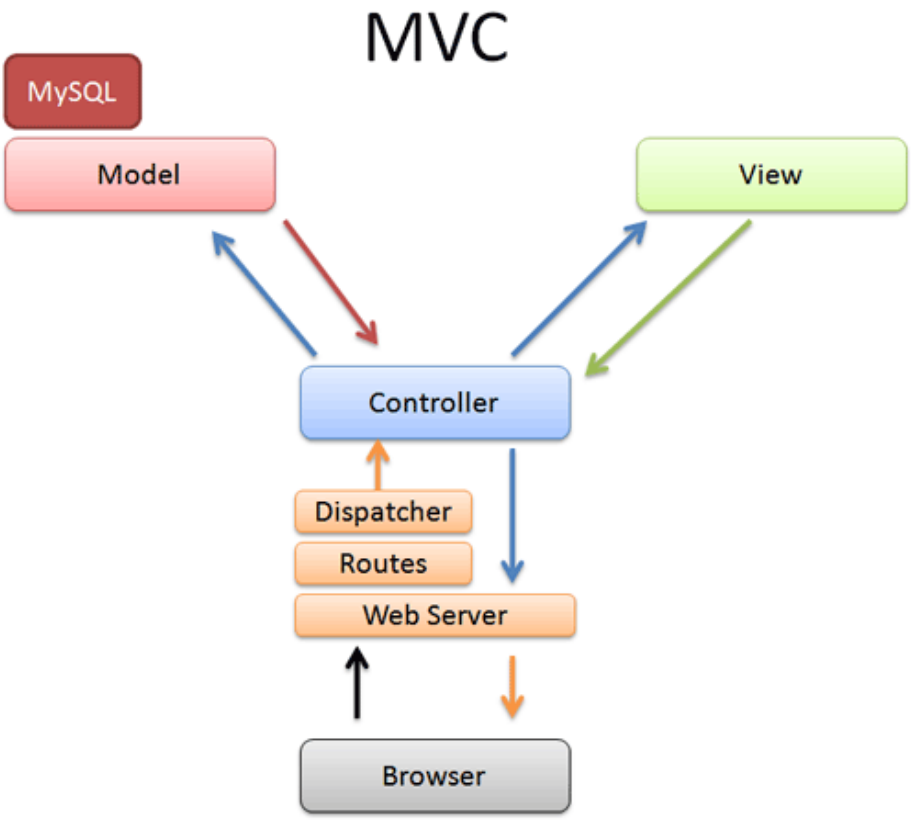

Figure 1: MVC Diagram

- **Typical Rails Architecture**

  The typical Ruby on Rails web application is based on the Model, View Controller (MVC) architecture. This architecture splits up the code of an app into three distinct part's, each responsible for their specific role in the apps functionality.

  When a web browser makes a request to a Rails application, first the web server will check if what has been requested is publicly and locally available. If it is,this is then returned without having to delve into the Rails application. An example of this would be a 404 page.

However, if the request cannot be completed by the web server, it will then hit the Rails router,which will then call on the corresponding controller action for the request. The controller is the "director" of the application,making decisions about the applications logic,requesting resources and handing them off to other parts of the app according to its called action.

The model is solely responsible for interacting with the application database and only interacts with the controller and the database. It stores and pulls data objects to and from the database in accordance with it's business logic. This differs from the application logic,as it sets the "rules" for how a database object will behave. The controller might submit a bunch of data to the model,which is then checked against this logic before being stored in the database, or the controller might request a specific object,which the model will then pull from the database and pass back to the controller.

The view is the part of the app responsible for formatting data into a human readable format. The view only communicates with the controller. Generally speaking, after requesting data objects from the model, the controller will then pass these objects or parts of these objects to the view. The view in turn will put these objects into a dynamically generated HTML document for example, but could also render a JSON,XML or some other human readable format. When this is done, it will hand the file back to the controller.

Finally, the controller will pass this completed view, which has been built with the model objects, according to the controller logic and the views rendering template, back to the browser to be displayed to the user.

Resources

- https://betterexplained.com/articles/intermediate-rails-understanding-models-views-and-controllers/

Q2

- **Common Database**

  PostgreSQL is a common database used in web applications. While it is currently one of the most popular database management systems (DBMS) in use today, there are also some specific drawbacks to it.

  Pro's

  - PostgreSQL is an older, open source DBMS. Due to this, it is compatible with many languages and many OS's such as Linux, Mac and Windows. Additionally, this means that PostgreSQL is also extremely stable and easy to integrate into new and existing projects.
  - As an open source DBMS, PostgreSQL is free to use for your application in whatever capacity you need it to operate in, no questions asked.

- As an open source DBMS, PostgreSQL has a large and diverse community of developers who are consistently contributing to the codebase.
- Again, tying into the above points, because of its age and stability, you can install PostgreSQL and, after some configuration, easily use it for simple operations without having to worry too much about maintenance.
- It is the closest DBMS to full SQL compliance.
- PostgreSQL is able to handle multiple users concurrently requesting and storing data at the same time. This means data is less likely to become corrupted during these operations and results in a much more robust web application.

Con's

- As an open source project, PostgreSQL is always changing. These changes must be documented but the standard of documentation is not consistent across the board. This essentially means you may spend a lot of time searching for documentation surrounding some of PostgreSQL darker corners.
- While PostgreSQL's ability to handle multiple users is one of its perks, the way it does this can become a con. When a connection is made, it is allocated approximately 10mb of memory. Naturally, this can add up quite quickly and is overkill for some operations.
- There are not as many tools available to aid with the management of a PostgreSQL database.

Resources

- https://www.digitalocean.com/community/tutorials/sqlite-vs-mysql-vs-postgresql-a-comparison-of-relational-database-management-systems

Q3

- **Agile Development**

  In understanding how to implement an Agile development methodology, first we need to understand the difference between an agile methodology and the Agile "philosophy" as a whole.

  The Agile development philosophy outlines broad ways of thinking about our projects and managing them. This is different to Agile methodologies, that build onto this principle and practical strategies for actually doing work in an Agile way.

  In this example, I will use the Scrum methodology to illustrate how a team would use Agile development for a project.

  1. Start with a strategy meeting. The idea behind this meeting is to layout the big picture and figure out what the core values of

the project/product will be. Additionally this is the point where there needs to be buy-in from all levels of the project, from client to management.

2. The next stage is to plan out the product backlog. This will be the "master" pool of work to get done which is consistently created and updated in conjunction with the client. The idea here is to plan out/update the goals for the project, rather than the specific features, and identify measurable ways of achieving said goals. This promotes a more agile mindset as it moves away from planning absolutely everything before the work begins and allows for flexibility in meeting the goal.

3. And just before the work begins, just like the product backlog, it can be worth thinking about what the release timeline will look like. This means deciding what you MVP will look like and where you want to take it from there on a loose "release by release" basis.

4. At this stage, it's time to sprint. A sprint is a period of time, between 2-4 weeks usually, where the team will actually try to implement some of the items in the product backlog by moving them into a spring backlog. Generally speaking, when the sprint has started, the team will work towards those goals and revise once the sprint has been completed. The sprint length will usually stay the same length for the duration of the project also. Towards the end of a sprint, the team will do a review and then a retrospective. The review is meant to be an evaluation of what has been done during the sprint, what seems like the next step and the product owner ultimately deciding if that iteration of the project will be released. The retrospective is more of an evaluation of the effectiveness of the sprint as a whole, looking at what worked well and what can be improved. It should also be mentioned that during this time their is consistent liaison between the developers and the client, in order to make sure their staying on target.

5. The team begins another sprint, creates another sprint backlog and the cycle begins again. This will continue until the end of the project of the needs of the project change. This could look like the final release of a product, or a massive change in product direction.

   Resources

   – https://plan.io/blog/ultimate-guide-to-implementing-agile-project-management-and-scrum/ https://www.atlassian.com/agile/scrum

Q4

- **Standard Source Control Workflow**

The goal of implementing an effective source control workflow is to mitigate the difficulty and risk in working on a codebase.

Developers will often have to work on portions of the same code-base simultaneously. Issues can arise when one developer unknowingly overwrites another developers code, or two developers work on separate version of the software but cannot make them compatible.

Using services, such as Git and Mercurial, developers now have a lot more flexibility in how they approach these sorts of issues.

A common strategy is to develop features using branches. Branches allow developers to create a "copy" of the trunk that they can work on independent to the main codebase. It should be noted that one of the advantages of using Git, is that when a project is branched, the codebase is not actually copied in its entirety. Git essentially uses "pointers" to keep track of what and what does not exist in a codebase at any point in time.

Usually a branch is created for each new feature or bugfix, worked on and then merged back into the trunk when it is ready. Developers can then evaluates the differences between the trunk and the branch if the trunk has changed since the initial branch before the final merge.

The upshot of this workflow is it's easier to keep the trunk version of the software stable and ready for release if it's not deployed already. This also makes it much easier to isolate the changes that happen to the trunk branch through the branches that were worked on.

A downside to this approach is that it can make testing more difficult if a developer branches a copy of the trunk, and then the trunk is further modified or added to after the branch has been created. This increases the likelihood of bugs once the branch is merged back into the trunk and increases the amount of time it takes to review the merge before it's committed.

Resources

- https://docs.microsoft.com/en-us/azure/devops/repos/git/git-branching-guidance?view=azure-devops
- https://dev.to/fpolster/how-to-work-with-git-an-overview-of-git-workflows-1icb

Q5

- **Automated Testing**

  Before or early on in a projects timeline, it is important to consider the role of testing. There are many different approaches to testing and no "right answer" as the needs of every project differs. Automated unit testing

is just one small way that a developer can test software in order to ensure it functions correctly.

Unit testing is testing small bits of code to ensure they are giving the desired output. For example, if you designed a function, ideally it would take an input, and give you some kind of output. Unit testing looks at these smaller, fundamental functions and matches expected outputs with actual outputs.

Using a test driven development (TDD) approach to create automated unit tests, the developer would first think about the function of the program and design the tests before the program itself. By using this approach, the developer intrinsically lays the foundation for a codebase that is more easily tested.

Once the test have been planned, the developer would then either create their own programmable tests or use third party software/programs to do so.

The next step is to actually create functions that pass these tests, and then refactor these functions into the wanted end program. Throughout the entire refactoring process the tests should always "pass". It is easy to imagine how much fore-thought needs to go into the program before a single line of code is written in order to create comprehensive TDD compliant unit tests for the lifetime of the project.

A huge advantage to using a TDD approach to create unit tests and then a program that will pass those tests is that it creates a robust code-base. If many of the smaller moving parts of a program are not intrinsically dependent on the whole program in order to function, they can be easily tested, debugged, modified and re-used elsewhere, potentially even in a different codebase.

Resources

- https://www.w3schools.in/software-testing/types/#Automation_Testing
- https://smartbear.com/learn/automated-testing/what-is-automated-testing/

Q6

- **Security Requirements**

  For a marketplace web-application based on the Rails framework there are some security requirements worth considering.

  If we think about what a marketplace needs from a user in order to function correctly, we can break the term "security requirements" down into "How do we protect users in these areas?"

Most marketplace apps rely on user to user interaction. Because of this, we need to differentiate our users, which in turns means storing some kind of information, unique to each user. This could be as simple as a password and username, to a full documentation of email, phone number and address. Regardless, the approach should be the same in that we will be storing data unique to a single individual, and that data should be protected and private from other users/attackers.

Additionally, sensitive credentials, such as passwords, phone numbers and addresses, could be encrypted to further protect user information from improper use. This would certainly be a requirement if user profiles contained any identifying information and is good practice in general for password storage regardless.

By the above argument, we can see that we will need to user a database system in order to store credentials, and ultimately, user products the web application. Therefore, we will be required to protect the database from SQL injection or other attacks that could compromise the database.

Another aspect for consideration is payment. Arguably the easiest way to "avoid" dealing with payment logistics is to have the two users sort out how they will arrange payment outside of the web application altogether.

Otherwise, it is imperative that we consider the PCI standards for payment processing when designing our application. This set of guiding principles ensures that payments systems are secure and are required if you want to conduct transactions using large companies products such as Visa and MasterCard.

Tying into this, if we are planning save user credit card information (if they elect to do so), this is very sensitive security area. Steps should be taken to ensure all data is encrypted and saved to the database in a way that cannot be easily accessed by developers or attackers.

User sessions will also have the be considered. As this is a marketplace, presumably users will be able to log-in in order to post things they wish to sell or log in in order to purchase items. This means a session token will be used. Because of this we will need to consider how we protect these session tokens from being discovered by attackers, which would allow them to totally skip a log-in and "impersonate" a user using their token.

Lastly, as a marketplace app where people can by and sell items, there needs to be some kind of monitoring around what is being sold. If the marketplace is for a particular kind of product, how are we going to ensure those are the products that are being sold. Additionally, how do we stop users from selling illegal items and filter out scammers.

Resources

- https://hackr.io/blog/mobile-app-security-standards-checklist

- https://www.lrswebsolutions.com/Blog/Posts/32/Learn-More/2018/11/11-Best-Practices-for-Developing-Secure-Web-Applications/blog-post/
- https://www.rsisecurity.com/compliance-advisory-services/pci/
- https://martinfowler.com/articles/web-security-basics.html

Q7

- **Common Methods for protecting information**

  I will address the points I made in questions 6 in this question and talk about methods for actually implementing the security relevant to this marketplace application.

  Taking information from a user and storing it in a database creates several security concerns.

  There is potential for SQL injection through a HTML form. The first line of defence here is using Rails validators. Rails validators allow us, in Ruby, to validate an input taken from a submitted form. The upshot of this approach is that the input can be checked before it hits a database or is potentially returned to a HTML/CSS/JS rendering browser. In this way we can safely handle potentially dangerous strings and reject them before they end up at their target destination in the framework.

  Its worth pointing out that if input doesn't pass validation, we want to reject that input and not render it back to the HTML document, as the string itself could be designed to target the front end encoding of the web application.

  Validation will be extremely important for our marketplace application, as users will often be interacting with different forms in order to create accounts, sell items, buy items and search for items. Using html elements to ensure the correct type of input is used for each field is one step that should be taken. Additionally, server side validation should take place in order reject or filter out any unwanted inputs. This can be done with rails using strong parameters, permitting only the required fields. This also has the upshot of intrinsically parameter binding our permitted parameters to make SQL injection almost impossible. Parameter binding is simply assigning the parameters to variables, then passing those variables to the SQL statement, rather than directly interpolating the potentially malicious strings into the statement.

  We can also control what hits the database by using validation in our models thanks to the Rails validate feature. With this we gain even further control over what hits our database in the first place. A form input that does not pass model validation will never interact with the database.

  Moving on, we can look at how we will protect the users data once it is in the actual database. It would be a breach of security and privacy if

a developer could simply interact with the back-end of the site and pull sensitive information, such as emails, passwords, addresses, phone numbers and any other personal information.

The first step is to ensure that the web application will have an SSL certificate when it is deployed. This will ensure that information sent between the client and server is secure, encrypted and unable to be intercepted by an attacker "listening in". Rails provides easy options for making production web applications use https (SSL compliant connections) by default.

The next step is to ensure that the information is encrypted when it is stored in the database.

Many 3rd party databases provide standing encryption. That is, the whole database is encrypted by default. However, this doesn't provide protection if an attacker gets hold of the database and is able to access it directly. The database will simply un-encrypt and serve up whatever is requested.

The way we can address this issue is by using application level encryption. This is where we encrypt the information before it hits the database, so if an attacker were to pull gain access to the database, they would simply get encrypted responses. There are a few ways to do this, and there's a many Ruby gems that allow us to easily encrypt and decrypt without directly interacting with the keys in a way that would compromise privacy. Making an educated decision about this should be a point of discussion during development.

Related to encryption and data transfer, ensuring transactions are safe and secure will be a big part of this application. An easier way to approach this is to use a third party payment processing service, such as Stripe or PayPal. This would require a secure redirect and return, to and from the third party processor. Given the resources needed to ensure that payments and payment details are not compromised or intercepted, it would be worth weighing the requirements and cost of having a PCI compliant system if one is not already in place with the cost of using a third party provider.

If PCI compliance is the goal, then heavy reference to the documentation will be needed to ensure that the overall system is safe from being compromised. This looks like ensuring that the system is fire-walled properly from any external source, and from internal sources also. SSL is an absolute must in this case as users will be sending credit card details between their client and your server. Encryption of these details if they are being stored is also mandatory. Regular testing and updating must also be a part of the strategy, as risks change and the system ages.

A marketplace app is likely to use session cookies in order to store the user state on a clients browser. This allows us easier way to create things like cart functionality and negates the need to re-authorise the user every time they make a new request. However, this does create a security concern.

These session cookies, if discovered by an attacker can lead to the session being "hijacked", with the attacker impersonating a valid users browser.

Having an SSL certificate and forcing the application to use HTTPS is a good start on how to keep attackers from intercepting a session token. Rails also has some great inbuilt security features for ensuring session fixation is impossible, such as resetting the session_id after each successful login. The Devise gem, a popular Rails gem for user login and management, incorporates many of these features by default.

It should be said that cookies should not be used by our marketplace application to store any sensitive data or "permanent" data. Session cookies are temporary in nature, and anything stored in them should also be fit the bill as temporary.

Lastly, the content on the marketplace should be monitored. A user should not be able to sell illegal goods through your marketplace and steps should be taken to try and identify and take down potential scams. This can be done using a web crawler that will alert an admin, or better yet, automate the process of removing potentially illegal content from the web application.

This should go hand in hand with a user policy, that if violated, will end up with the user being banned from the platform. Additionally, steps should be taken to increase awareness of other users around scams and ensuring they have some degree of trust in the seller. This can be potentially be achieved through a vouching system, where sellers with a good track record will get a rating, etc.

Going hand in hand with this awareness, allowing users to report suspicious or illegal content can help fill in the gaps of a web-crawler, which might miss a more nuanced attempt at selling illegal products.

Resources

- https://martinfowler.com/articles/web-security-basics.html
- https://ankane.org/sensitive-data-rails
- https://www.netsparker.com/blog/web-security/definitive-pci-dss-compliance-guide-web-application-security/
- https://guides.rubyonrails.org/security.html
- https://evercompliant.com/fraud-prevention-tool-box-content-monitoring-on-steroids/

Q8

- **Legal Obligations**

  The first step in determining what the legal obligations are is determining whether ACME corp is required to comply with the Privacy Act. If the business turns over more than three million dollars a year, then they are required by Australian law to comply.

It should be pointed out that even if ACME doesn't fall within this requirement for compliance, following the principals anyway would be a good idea to protect from other unlawful breaches of privacy and gain user trust.

The Australian government has laid out the Australian Privacy Principles (APP) for aiding businesses in knowing what their obligations are under law. There are 13 principles in all.

1. APP compliant Privacy Policy. The company must make available to users a privacy policy that contains information regarding their person information. The policy should contain:
   - What kind of information is stored and why.
   - How someone can access their personal information and change it.
   - How to lodge a complaint.
   - Information around how the business will use the information and whether it will be disclosed to other entities.
2. A user should be able to be anonymous or use a pseudonym in place of anything that could be potentially identifying
3. The company should not store or collect personal information that is not required for functionality of the application.
4. The company should have a policy for dealing with personal information that is voluntarily given. For example, if information is given to the company without any collection on the companies part, and that information is not publicly available, the information should be destroyed.
5. The company must notify the user that they are collecting personal information at the time of collection and inform them for what purpose.
6. The company must tell the user under what circumstances that it would disclose the personal information of the user to an external entity.
7. The company must not disclose user information for direct marketing purposes unless that is what is expected of the company.
8. The company should ensure that any overseas recipient of user information falls in line with the APP principles also.
9. The company should not identify itself as government affiliated in anyway, unless it is.
10. The company should ensure that the user information is up do date and complete.
11. The company must take precautionary steps to make sure the data is protected and cannot be maliciously accessed.
12. The company must provide the personal information to the user who's information it is if a request is made to do so.
13. The company must request the user to update their information if

they believe it is inaccurate or out of date.

Resources

- https://gbksoft.com/blog/legal-pitfalls-of-app-development/
- https://legal123.com.au/how-to-guide/how-to-develop-an-app-infographic/
- https://www.oaic.gov.au/privacy/guidance-and-advice/mobile-privacy-a-better-practice-guide-for-mobile-app-developers/
- https://www.oaic.gov.au/privacy/the-privacy-act/rights-and-responsibilities/
- https://www.business.gov.au/Risk-management/Cyber-security/How-to-protect-your-customers-information
- https://www.oaic.gov.au/privacy/australian-privacy-principles

Q9

- **Relational Database Structure**

  A relational database structure is based on "tables". Each of these tables within the database has a unique name, generally specifying its purpose. The table itself contains columns which generally specify the name of the data that is being held there. For example, a "user" table may have a column for name, email, age etc. These are essentially keys to the data that is contained within the table.

  Its also worth mentioning that in a relational database, all tables will have a primary key, which is a way of identifying each individual entry on a table.

  Once the columns of a named table are defined, we add data to the table in the form of rows. Every time a new row is added, it is assigned a unique primary key and then the rest of the now created cells are now filled out accordingly.

  These columns can also reference foreign primary keys, and the database will know which table to look to for the primary key. For example, if a "post" table had a foreign "user_id" key, it would know to look at the user table in order to locate that primary key.

  This is a key part of relational database structure, as the structure itself assumes that data that is stored will relate specifically back to something in particular.

  Resources

  - https://www.relationaldbdesign.com/database-design/module2/intro-relational-database-structure.php

Q10

- **Relational Database Integrity**

  Relational databases creates integrity in the way it stores information intrinsically by providing each stored piece of information with its own primary key. When this happens it makes sure that there is no duplicate data within the database. You could create a table and fill out the columns identically for every entry, but the primary key will always be different.

  This also creates integrity around the relationships within that database. Using foreign keys, a database table will know exactly where and what to look for at all times.

  Relational databases also require specification of the data type being entered, and this can be further improved upon using application level validation to ensure that the data that is expected in each column is indeed correct.

  Resources

  – https://www.talend.com/resources/what-is-data-integrity/

Q11

- **Relational Database Manipulation**

  Relational databases generally use Structured Query Language in order to store and manipulate information.

  This language allows us to essentially perform relational algebra on a data base in order to select or add the information we want to it without getting bogged down in mathematical terminology and notation.

  This allows applications and developers to use powerful statements to not just search through databases, but also to insert and delete information from these databases using fairly semantic inputs. SQL itself is built around the relational database model, so its statements are structured to comply to the relational nature of the database.

  For example, if we wanted to insert information into a table named “users”, and the fields we want to complete are “name” and “email”, we could use the following SQL statement:

  ```
  INSERT INTO users (name, email) VALUES ('Sam', 'sam@sam.com');
  ```

  This statement highlights the nature of manipulation in the database. To further highlight this, we could recall the entire row of data using a statement such as:

  ```
  SELECT * FROM users WHERE email = "sam@sam.com";
  ```

This would return all columns from all entries where email matches the query string.

Relational databases also intrinsically look through tables that are referenced in the statement.

Resources

- https://en.wikipedia.org/wiki/Data_manipulation_language

Q12

- **Sorting Algorithms**
  - Selection Sort

    A selection sorting algorithm is a simple sorting algorithm used to sort an array in ascending order.

    The basic idea behind a selection sort algorithm is that it will iterate through an array and select the lowest number from the array and place that into a sorted array. It will continue to do this until all numbers have been sorted in order into the new array.

    This can be done by creating a brand new array, or partitioning the array in-situ and swapping variables.

    I will describe the in-situ variant.

    We start by selecting the first element of the array. I will call this the sorting index. This intrinsically also becomes our current minimum. Next we look at the following element. If this element is smaller than the current minimum, it becomes our new minimum. We then evaluate the following element and do the same process, moving our current minimum when a smaller value is encountered.

    When we reach the end of our array, we swap the current minimum with the sorting index. We now consider the number that has been swapped into the sorting index as part of the sorted partition. As we go on, we will have the sorted partition left of the sorting index and the unsorted partition to the right.

    We continue this process until the algorithm iterates through the entire array.

    Below I have included a code example of this algorithm in Ruby.

```ruby
def selection_sort(array)
  # First we create a variable to store which position we are swapping
  sorting_position = 0

  # We create a loop that runs as many times as the size of the array in order to s
```

```
  array.size.times do

    # We set the current minimum to the position we're sorting from
    current_minimum = sorting_position
    # We set the current item to the position one ahead of of the current minimum
    current_item = sorting_position + 1

    # We nest a loop to iterate through the unsorted part of our array
    (array.size - (sorting_position + 1)).times do

      # We compare our current minimum with our current item
      if array[current_minimum] > array[current_item]
        #If the current minimum is larger than the current item, we set the positio
        current_minimum = current_item
        #We then increment the current item position var by one
        current_item += 1
      else
        #if the current minimum is smaller than the current item, we increment the
        current_item += 1
      end
    end

    # We then swap the position of the value at the sorting position with the value
    sorting_position_value = array[sorting_position]
    array[sorting_position] = array[current_minimum]
    array[current_minimum] = sorting_position_value

    # We then increment our sorting position by one to create our sorted partition
    sorting_position += 1

    #The loop then continues until is has sorted all the elements into the sorted p
  end

  return array
end
```

– Counting Sort

The counting sort is interesting in that it can perform faster than most sorting methods if the right conditions are present. It should be noted that the counting sort is only an option for arrays of integers.

The speed counting sort relies on two integral things. The range of an array must be known and the range should not be drastically larger than the number of elements in the array. For example, a 5 element array with a range of 1 to 10,000 will run much slower than an 100 element array with an range of 100.

The basic premise behind a counting sort is that it creates another array (or hash) that is responsible for counting the number of times a particular value appears in an array. First we create an array or hash with the same range as the array to be sorted. For this example I will use a hash.

We then iterate over the array to be sorted and increment the count on each corresponding hash key by one until we reach the end of the input array. We now have a hash of integer keys that point to the amount of times they occur in the input array.

Now we can iterate through our counting hash and input the key into a new hash corresponding to the value. For example, if the key "3" has a value of 7, we would push 3 onto the new array 7 times. We continue doing this until we reach the end of our counting hash.

We now have a new array with the sorted values of the old array.

There are many different ways to do this that are more space efficient and use only two arrays rather than an array and a hash.

```
def counting_sort(arr,range)
#We generate a hash with with keys that correspond to every number within the arr
setup = *(1..range)
count = Hash[setup.collect { |value| [value, 0] }]
#We then generate a new array to push our counted hash into later
finalarr = Array.new

#We iterate through the array and increment the count on the hash where there is
arr.each do |element|
  count[element] += 1
end

#We then iterate through the hash and look for keys with values more than zero
count.each do |key, value|
  if value > 0
    #if a key has a value more than zero, we push that key to the new array as mo
    value.times do
      finalarr << key
    end

  else
    #Otherwise, we move onto the next key
    next
  end
end

return finalarr
```

```
end
```

In terms of Big-O notation, we can see that the selection sort method uses two loops to achieve the desired output, one nested inside of another.

If there was only one loop, then the complexity would grow linearly. This basically means if we asked a program to iterate over an array of 10 elements, it would take 10 times the amount of time to do the same operation over a list of 100 elements. We can say that the complexity of the method grows linearly with the amount of data points given to it. This is expressed as O(n).

But, as previously mentioned, the above method uses a nested loop. In this particular example the program will have to iterate over the array, eg O(n), but within that iteration, another iteration occurs, eg another O(n). This gives us O(n2), or the square of N is now the complexity. So to use our previous example, an array of 10 elements is no longer O(10), but O(10*10). Now our array of 10 elements takes as much time to process as our array of 100 elements, and our array of 100 elements takes 100 times longer than it did previously.

As we can see, these numbers grow quite quickly with larger datasets.

The counting sort provides an interesting example of Big-O notation. Technically, the complexity of the counting sort above is O(n). However, the size of this N value is dependant on the size of the range of the array to be sorted.

We can express this more clearly by writing O(n + k), where n is the size of the input array and k is the size of the range. While this complexity is linear, if we had a very large range, like 10,000, and a small array, like 5, we're iterating through a large number of keys in a hash for only a small amount of elements to sort.

Within the above example it should also be noted that there is technically a nested loop that executes every time a value more than 0 is found in the count hash. This does not add to the Big-O complexity however, as this times loop will only ever execute as many times as the input array is large, instead of fully looping through the input array each time it is called. It therefore does not make the notation O(n2)

Where the counting sort really shines is where we know the range will be close to, or less than the size of the array to be sorted.

To comparing these methods is a matter of context. If we have an array where the range of the integers to be sorted is known and close to or less than the number of elements in the array, the counting sort is superior.

For example: "‘ Array Size = 50, Range = 70
=> Counting Sort = O(120) => Selection Sort = O(2500)

Array Size = 50, Range = 5,000 => Counting Sort = O(5,050) => Selection Sort = O(2500) "‘

We can see here, according to context, that the counting sort works better in the correct context; with integers and when the range is known.

Resources

- https://gist.github.com/brianstorti/953310
- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- https://medium.com/basecs/counting-linearly-with-counting-sort-cd8516ae09b3

Q13

- **Search Algorithms**
  - Linear search A linear search is as simple as a search algorithm can get. A linear search will take a sorted or unsorted array and a value that it is looking for. It will then iterate over the array and compare the value its looking for with each value in the array.

    ```
    def linear_search(array, find)
    # We iterate through each element of the array
    array.each do |element|
      # We compare each element to the value we're looking for
      if element == find
        return "#{element} found"
      # If it doesn't exist, we move onto the next element
      else
        next
      end
    end
    # If the element is not found after a complete iteration over the array, we retu
    return "#{find} not found"
    end
    ```

    This extremely simple approach is not complicated and has a fairly low complexity, but even at this complexity, very large datasets can take a long time to iterate through large datasets. This is where we can use a binary search in the right context to make the search faster.

  - Binary search

    Binary searches are a faster way of performing a search for a specific index over a large dataset. The one catch to a binary search is that it must be performed on a sorted dataset.

    The basic functioning behind a binary search is that it will take a sorted array, then look at the midpoint of that array and see if it matches the value that we're trying to find. If the element at the mid index is higher than the value we're searching for, we throw away the

top half of the array, if it's lower, then we throw away the bottom half.

With this new array, we then repeat the process, looking at the midpoint, checking if it matches, and discarding the upper or lower half of the array accordingly. This process continues until we find the value we're looking for, or reach an array of 1 element that does not match, in which case the value we're looking for does not exist within the array.

```ruby
#We set default values for the minimum index and max index
def binary_search(sorted_array, find, min=0, max=sorted_array.size-1)

  #We create a midpoint var to help us point to the middle of the array as define
  midpoint = (min + max) / 2

  #We use a spaceship operator to compare the midpoint element with the value we
  case sorted_array[midpoint] <=> find
  when 0
    #We return that the element has been found
    return "The element is at index #{midpoint}"
  when -1
    #We check if there is only one element left in the array and halt the method
    return "Element not present" if (min - max) == 0
    #Otherwise, we set our minimum index to 1 above the midpoint
    min = midpoint + 1
    #We the recursively call the method on itself with the updated min index
    binary_search(sorted_array, find, min, max)
  when 1
    #We check if there is only one element left in the array and halt the method
    return "Element not present" if (min + max) == 1
    #Otherwise, we set our maximum index to 1 below the midpoint
    max = midpoint - 1
    #We then recursively call the method on itself with the updated max index
    binary_search(sorted_array, find, min, max)
  end
end
```

In terms of Big-O notation, both of these search methods are relatively low complexity, especially when contrasted with the sorting algorithms. However there are differences between the two search methods described above.

The linear search has a Big-O complexity of O(n), so the complexity increases linearly with the number of elements given to it. It will take 100 times longer to iterate over 1,000 elements as it would 10. As we can see,

this is fine up to a point, but if we're working with large datasets this can become an issue.

This is where the binary search can shine. As long as the array is sorted then the binary search has a complexity of O(log n). log n is basically the opposite to exponential growth. Exponential means that with every operation our workload increases by double or more. Logarithmic is the opposite, meaning our workload will be at least halved on each operation. Looking at the code above, we can see that the binary search function quite literally halves our array each time it performs our method. This results in a logarithmic runtime, and can be easily visualised as a curve on a graph that starts steep, then flattens out as time goes on.

With large datasets, this can be powerful. If we have a dataset of 10,000 data points, and we double this to 20,000, a O(log n) function like a binary search will only take one extra operation to reach the same complexity as the same method run with 10,000 data points. Very cool.

Resources

- https://rob-bell.net/2009/06/a-beginners-guide-to-big-o-notation/
- https://medium.com/better-programming/a-gentle-explanation-of-logarithmic-time-complexity-79842728a702

Q14

Airbnb

1. **Describe the software used by the app**

   Airbnb uses a huge range of software to deliver its two sided marketplace. I will touch on the biggest and most relevant pieces of its software stack:

   - **Ruby on Rails:** The base framework that Airbnb is built on is Ruby on Rails. Rails is written and operates in Ruby. However, Rails simply handles the backend for airbnb, whereas the front end is handled by React, described below.
   - **React:** React is a popular javascript framework used mainly for front end design. This framework is built on the Javascript language. In this case, combining React with Rails gives Airbnb the flexibility in front end design that React caters for, while maintaining a Rails backend.
   - **Nginx:** Nginx is a webserver specifically designed to be scalable and flexible under high loads. This webserver is what handles all the http requests of browsers trying to connect to parts of the web application and interacts with the app in order to deliver the desired content.
   - **Redis:** Redis is, simply put, a non-relational "caching" database. Redis acts as a temporary place of storage for simple kinds of data. It's

purpose is to reduce the amount of calls made to the main database by temporarily holding onto data called from the database, therefore improving speed and potentially decreasing cost.

- **Amazon S3:** The Amazon S3 database is used to store images and media for the airbnb site.
- **Amazon EC2:** The Amazon EC2 acts as a cloud host that can respond to the demand of Airbnb traffic. It also means that if Airbnb goes down somewhere, EC2 can take on the extra workload while the problem is fixed.
- **Amazon RDS:** The Amazon RDS acts as the main and centralised relational database for the Airbnb app.
- **Hadoop:** Hadoop is a software tool that "splits up" the work of making database queries over more than one computer. Simply put, if one computer makes one large request to the database, the database must send back all of that data to the same computer. But if several computers make smaller requests to the database, the database can deliver all these parts simultaneously to different computers, which then reduce these smaller parts into one larger part, resulting in a faster, more efficient request.
- **Airflow:** Airflow is a simple tool that allows tasks to be run in order of operations, automatically. Say for example, you wanted to generate an email, but needed to collect a bunch of data in order to do this properly, and the email itself may look different depending on the state of whatever you're checking it against. Airflow can move through the application and piece these things together once the workflow has been designed for it.
- **Druid:** Druid acts as a big data synthesizer, that can listen in on events of the app and aggregate these into statistics. These numbers can be used hand in hand with creating user interfaces relating to these values.
- **Twilio SendGrid:** Twilio is an API used for sending emails to users.
- **Braintree:** Braintree is a online payment processor, like Stripe, that allows apps to take payments securely.

2. **Describe the hardware used to host the app**

   The entirety of Airbnb's physical hosting is done through Amazon EC2 server hosting. This means the physical hosting of the application is all done via the cloud and is entirely scalable to the needs of Airbnb at any given time. When there is more demand for the application, EC2 simply responds by using more server to handle the load. When the load decreases, those servers stop hosting.

Airbnb also uses Amazon S3 and RDS for its data hosting, which again, allows it to scale its cloud based hardware with its needs.

The huge upshot of using a cloud based system is that it cuts out a huge amount of work for a company if they don't have to worry about storing and maintaining servers. The downside here is that as the application grows and demands more resources, the cost of these resources intrinsically increases.

3. **Describe the interaction of technologies within the app**

   Ruby on Rails acts as the base framework for the airbnb application. This handles all the back end processing of the application. The is integrated with React in order to create a complex front end that is able to dynamically render content depending on the browser state. All of this is served through the Nginx webserver which handles the connections and requests from browsers asking for data from the application. When a request is made to the application through the webserver and a database query needs to be made, this is done through an interplay of a few different technologies. For images, the application will query the S3 database, which is responsible for cloud storage of media. For other database queries, the application will interrogate the RDS, Amazons relational SQL database, using Hadoop, to split the requests over multiple ports of call and then reduce them into the final data object, to complete the request faster. At this stage, what can be cached in Redis, which is responsible for temporarily holding onto specific data types in a non-relational way, will be cached, so less successive database queries need to be made if the browser returns to an old page, or a new page requires the same data as a previous one. Running in parallel with all of this are a few technologies also. Airflow, responsible for workflow automation, will be working in the background, automating tasks that must be done according to a workflow, such as email, which is then handled by the Twilio SendGrid API. Druid is also listening in to the events on the application and presumably having that data synthesized meaningfully elsewhere. Braintree is the payment processor that steps in when a payment action is required on the site.

   This is all hosted and scaled on AWS services, which look after the maintenance and scaling of the application, leaving the application team to solely focus on the interplay of the apps technologies without worrying about the size or speed limitations of the hardware needed.

4. **Describe the way data is structured within the app**

   Airbnb is a two way marketplace, which means its data structure will generally conform to a few concepts. In a two way marketplace, there are two users, one is a buyer and one is a seller. These users can be fundamentally differentiated in how they are regarded as data, or every user can be both. In the case of Airbnb, every user can be both a buyer and a seller.

A point of user data will generally hold on to all the details unique and specific to that user, as well as pointing to anything that may relate to that user, whether it be places they're hosting from, experiences they're delivering or simply favorites they've saved to look at later.

A marketplace generally has some kind of product that is being sold by users. In the case of airbnb, there are two fundamental products that the application caters to; accommodation and experiences. Users can both create and sell these things as well as buy/book these things from other users.

Both of these points of data would hold on to a bit of data unique to themselves, like name and description, while potentially pointing to other things such as reviews, category, amenities, features and ratings which may exist over more than one place of experience.

In order to handle the actual booking of these places/experiences, there would also need to be a data structure that handles the booking that can be referred to for the seller, the buyer and others interested in the product.

Reviews would be a part of the data structure also, and can be associated with any of the above parts of the structure depending on the context. For example, a review might be a user review, or an experience review, etc.

The inbuilt messaging tool also is a part of the data structure that needs to be considered. Users can have multiple conversations with unique messages in each that are attributed to a unique user.

5. **Tracked entities**

   For brevity I have only mapped the airbnb marketplace for the interaction of users and rooms, omitting experiences. I understand that experiences operate in much a similar way to rooms, but I think that demonstrating the same thing twice is unnecessary and would sacrifice readability.

   - **Users:** Stores all unique information relevant to a specific user, such as name, password, email and other personal details. Also contains id numbers for other services such as google and facebook and boolean values regarding the users particular binary settings.
   - **User reviews:** Contains a short review of a particular user, from another user
   - **User review comments:** a responding comment of a user review, from another user.
   - **Card details:** Contains all the card details of a particular users card and its country
   - **Country:** Contains all countries

- **PayPal's:** Contains all PayPal details, each specific to a different user.
- **Australian Bank Details:** Contains all bank details, each specific to a different user.
- **VAT id's:** Contains VAT tax codes, each specific to a different user.
- **Notification preferences:** A series of entities that contain boolean operators regarding if a particular user wishes to be notified and in which ways they want to be notified. The entities are separate as the same way of communicating can be used for different types of notification, for example, account vs policy notifications via text.
- **Languages:** Contains all spoken languages.
- **User languages:** A joining entity between languages and users.
- **Emergency Contacts:** Contains contact and relationship information about a specific individual and which user they relate to.
- **Conversations:** Contains the information regarding a particular booking and the two users interacting
- **Message:** Contains a message as well as the user who sent it and what conversation it is a part of.
- **Saves:** A list of favorites, or saves, that contains information about the room saved and the user who saved it.
- **Genders:** A simple entity to store different genders assignable to a user.
- **Site languages:** Contains all the name of languages that the site can be displayed in.
- **Currencies:** Contains all the available currencies for transaction on Airbnb.
- **Timezones:** Contains all the timezones.
- **Rooms:** Contains information specific to a room, such as the host, its rating, prices and fees, its location, number of beds, bathrooms and its description.
- **Room Categories**: Contains a list of possible room categories.
- **Room cancellation policies:** Contains information on a cancellation policy, like how many hours for a free cancel, what window there is for a partial refund, what percentage is refunded, and whether a service fee is charged.
- **Check-in types:** A list of different types of check-in for rooms.

- **Room discount policies:** Contains information as to whether a room has a weekly or monthly discount, and if so, what the percentages of those discounts are.
- **Room pictures:** Simply contains a bunch of pointers to amazon s3 images for a particular room.
- **Room amenities:** Is a joining entity for rooms and amenities
- **Amenities:** Contains a list of amenities.
- **Room reviews:** Contains information about a room and a user who stayed their as well as a review.
- **Ratings:** Contains a set of values relating to a room review around the different aspects of a room.
- **Bookings:** Contains information about the user booking, the room to be booked, the dates of booking and amount of nights, as well as if the booking has been cancelled and what that cancellation cut-off is.
- **Transactions:** Contains information about the booking it relates to, as well as the payer and payee information, the currency to be paid, the price, discount, total and deadline for this payment. It also tracks whether the payment is outstanding and whether its a refund.

6. **Entity relationships**
   - **Users:**

     Has one:

     - Gender
     - Site language
     - Currency
     - Timezone
     - PayPal
     - Bank details
     - VAT
     - Notification Preferences

     Has many:

     - Languages THROUGH User Languages
     - Emergency contacts
     - Messages
     - Conversations
     - Rooms
     - Bookings
     - Transactions
     - Card Details
     - Saves

  - Room Reviews
  - User Reviews
  - User Review Comments
  - Emergency Contacts
- **Notifications Preferences:**

  Belongs to:

  - User
- **VATS:**

  Belongs to:

  - User
- **Bank Details:**

  Belongs to:

  - User
- **PayPal's:**

  Belongs to:

  - User
- **Card Details:**

  Belongs to:

  - User

  Has one:

  - Country
- **Countries:**

  Belongs to:

  - Card details
- **Saves:**

  Belongs to:

  - User
  - Room
- **User Reviews:**

  Belongs to:

  - Reviewer
  - Reviewee

Has many:

- User Review Comments

- **User Review Comments:**

  Belongs to:

  - User Review

  - **Emergency contact:**

  Belongs to:

  - User

  Has one:

  - Language

- **User Languages:**

  Has many:

  - Users
  - Languages

- **Messages:**

  Belongs to:

  - Conversation

- **Conversations:**

  Belongs to:

  - Host
  - Client
  - Booking

  Has many:

  - Messages

- **Rooms:**

  Belongs to:

  - User

  Has one:

  - Room Category
  - Room Cancellation Policy
  - Check in type
  - Discount Policy

  Has many:

  - Room pictures
  - Amenities THROUGH Room Amenities
  - Room Reviews
  - Bookings
  - Saves
- **Room Categories:**

  Has many:

  - Rooms
- **Check in type:**

  Has many:

  - Rooms
- **Room Cancellation Policy:**

  Belongs to:

  - Room
- **Discount Policy:**

  Belongs to:

  - Room
- **Room Pictures:**

  Belongs to:

  - Room
- **Room Amenities:**

  Has many:

  - Rooms
  - Amenities
- **Amenities:**

  Has many:

  - Rooms THROUGH Room Amenities
- **Room Reviews:**

  Belongs to:

  - Room
  - User

  Has one:

  - Rating

- **Ratings:**

  Belongs to:

  - Room review

- **Booking:**

  Belongs to:

  - Room

  Has one:

  - User
  - Conversation

  Has many:

  - Transactions

- **Transactions:**

  Belongs to:

  - Booking
  - Payer
  - Payee

  Has one:

  - Currency

7. **Airbnb database schema**

Resources

- https://www.airbnb.com.au
  - https://stackshare.io/companies/airbnb
  - https://www.forbes.com/sites/quora/2018/02/20/what-technology-stack-does-airbnb-use/#286998c64025
  - https://www.nginx.com/resources/wiki/
  - https://redislabs.com/ebook/part-1-getting-started/chapter-1-getting-to-know-redis/1-1-what-is-redis/
  - https://www.youtube.com/watch?v=9s-vSeWej1U
  - https://www.youtube.com/watch?v=xQBNn67rL_o

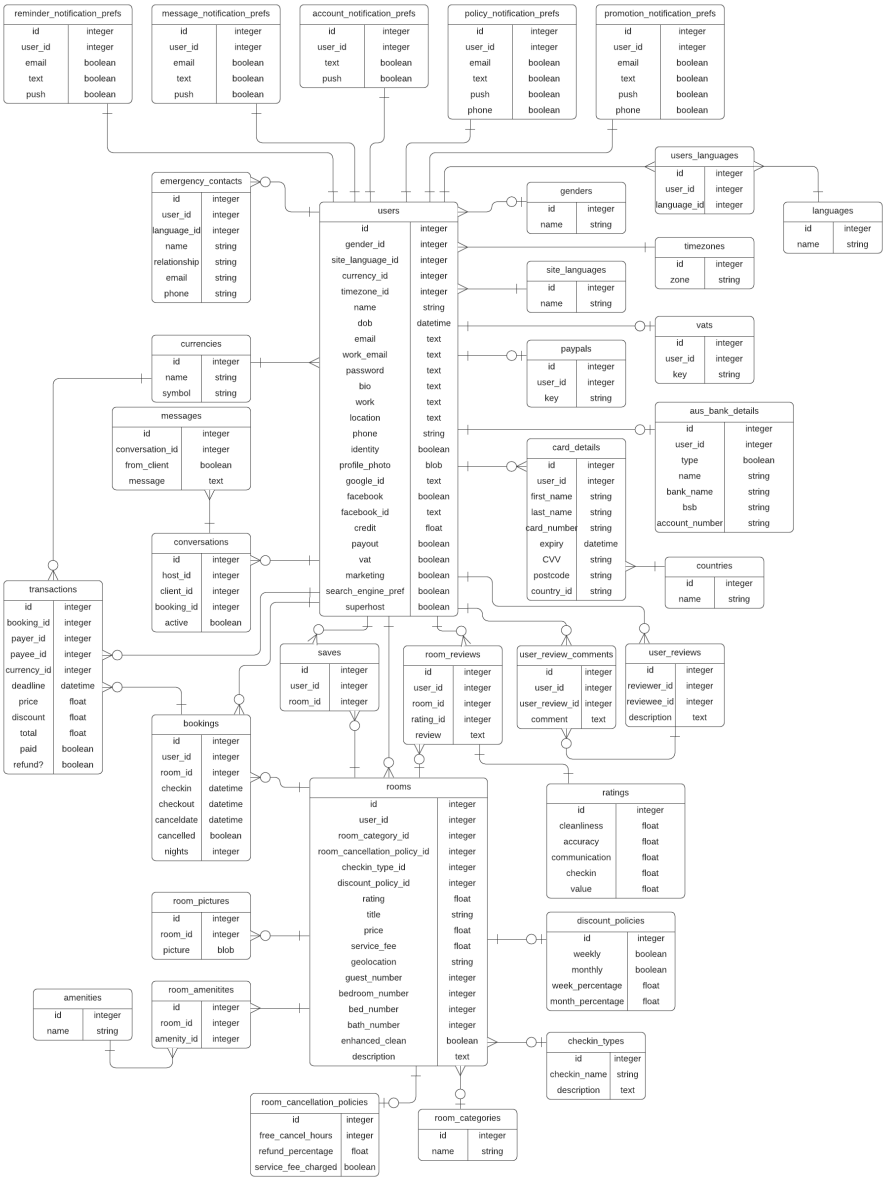

Figure 2: airbnb ERD