DevOps (contd.)

Database Optimization

N + 1 Query Problem

- Issue: do a query and for each result of that query, do another query on that
  - $O(mn)$ time and space
- Can use `includes` function on an active record collection to include the conditions for the join in the collection returned

Defending Customer Data

SSL / TLS

- Idea: encrypt HTTP traffic, foil eavesdroppers
- Uses Public Key Cryptography to share a symmetric key
- Can add `force_ssl` in Application Controller to force some or all actions to use SSL
  - In practice, causes an HTTP redirect to the SSL version
- No assurances of who sent messages or that the data is secure

SQL Injection

- Can use built in sanitization
  - eg: `Moviegoer.where("name=?", params[:name])`
- Can use dictionary style params
  - eg: `Moviegoer.where(:name => params[:name])`

XSRF/CSRF

- Makes a request that harvests a cookie and can then be used to impersonate the requested
- Referrer is not good enough because it can be spoofed
- Can't trust anything that comes from the user
- Can set `SameSite` cookie to `strict` to ensure that the cookie is only included if the request is coming from the same site
- Can use a nonce (in rails with `< %= csrf_meta_tags %>` and `protect_from_forgery` in `ApplicationController`) which makes sure the requests are responses from the most recent request
  - Also prevents replay attack

DevOps Fallacies, Pitfalls, and Concluding Remarks

- Avoid premature/unwise optimization
- Speed is a feature $\triangleq$ monitoring is your friend
- Horizontal scaling » per-machine performance
- Design to avoid terrible performance vs worry about optimal performance
- DB is particularly hard to scale
- Cache at many levels
- Use PaaS for as long as you can
- Security is hard to add after the fact
- Stay current with best practices and tools
- Prepare for catastrophe $\triangleq$ keep regular backups of site and db

Legacy Code

- Still meets customer needs

- You didn't write it, and it's poorly documented
- You did write it, but a long time ago and it's poorly documented
- It lacks good tests (regardless of who wrote it)

Ways to Modify Legacy Code

1. Edit and Pray $\triangleq$ 'I kind of think I probably didn't break anything'
2. Cover and Modify $\triangleq$ let test coverage be your safety blanket

- Exploration $\triangleq$ determine where you need to make changes (change points)
- Refactoring $\triangleq$ is the code around change points (a) tested and (b) testable
  - Just b $\triangleq$ improve test coverage
  - Neither a nor b $\triangleq$ refactor

Working With Legacy code

1. Get code running
   - Good to create a scratch branch that is never checked into VC
   - Learn the user stores $\triangleq$ get customers talk you through what they're doing
2. Discover the domain model
   - Understand db schema
   - Create a domain-model or entity-relationship diagram if possible
   - Consider the most important (highly-connected) classes, their responsibilities, and their collaborators
3. Run tests, and read main integration tests if you have them
   - Can check stats with `rake stats` for code to test ratio
4. Reference informal design docs
   - Mockups and user stories
   - Archived email, wiki, shared drive
   - Project software (if possible)
5. Keep expanding design docs as you go
6. Attempt to run embedded documentation parsers if possible