How to build a MVP web app

Minimum Viable Product

http://leanstack.com/minimum-viable-product/

Iterative Development

https://m.youtube.com/watch?v=feKv4wheDeU

Continuous Delivery

https://en.wikipedia.org/wiki/Continuous_delivery

http://www.thoughtworks.com/continuous-delivery

http://martinfowler.com/books/continuousDelivery.html

Delivering value from day one

Deployment first:

Deploy a production grade hello world app from day one
Deploy early - deploy often
Deliver small incremental features that deliver increasing value

Robust, Reliable, Professional
Build Half a product, not a half-assed product

https://gettingreal.37signals.com/ch05_Half_Not_Half_Assed.php

Networking and Security:
Point a real domain to application from day one
Install SSL certs.(Dev\Prod mode switch)

Background tasks(optional):
Setup background tasks
Setup email delivery system

Errors and Maintenance:
Error pages
Maintenance page
Read only mode controls

Testing
Endpoint (URL, Route) handler tests
Browser compatibility tests (SaaS tools)

Branding & legal:

Logo (text based is OK)
Favicon
Tagline (your hello world)
Privacy,Terms,Copyright
About page (why,who,contact info)

Customer relations:

Customer feedback inbound email
Customer support inbound email
Company blog
Technical blog
Twitter account for site status

Collecting Feedback

Monitoring
Alerting
Analytics
logging
Error tracking
User Activity tracking

Collecting Feedback

What is your app doing?
Is it up?
Do all end points work?
How many users are on?
How many were on yesterday?
What did they do yesterday?
How many errors occurred?
What were those errors?
Are there slow pages?

Collecting Feedback

Add log sink

Add Performance monitoring and alerting services

Add Error monitoring and alerting services

Add metrics collection code and reporting

Add Membership system with user activity tracking

12 factor apps

http://12factor.net/

http://12factor.net/dev-prod-parity

http://12factor.net/config

Development Production Parity

Working database Backups and restores:
today you should be working with production data from last nights backup

background process development sandbox

application configuration with environment variables

use same js\css content pipeline for dev and production with source maps

Dev Prod Parity Tooling

Forman proc file process manager(Heroku)

.env file based environment variables(Heroku)

database backup\restore to AWS S3

development SMTP server

Vagrant, Docker, etc

Optimizing for speed of iteration

Reduce the impact of code change

Reduce moving parts

Automate all the things

documented workflows

http://www.slideshare.net/kellan/architecting-for-change-qconnyc-2012

Continuous integration

http://www.thoughtworks.com/continuous-integration

Cloud Hosted:
Travic-ci, Circle-ci, codeship, etc

Self Hosted:
jenkins, bamboo, etc

Feature Flags

http://blog.travis-ci.com/2014-03-04-use-feature-flags-to-ship-changes-with-confidence/

http://martinfowler.com/bliki/FeatureToggle.html

Caveat

http://swreflections.blogspot.com/2014/08/feature-toggles-are-one-of-worst-kinds.html

Instant Code Rollbacks

Rollback to any previous version of application

Break before Make Database migrations

1-add new tables and\or columns
2-deploy new code
3-remove unused tables and\or columns

Embrace coding anti-patterns

This is going to be controversial!

(Avoid) Separation of Concerns

write less modules, classes, functions and methods

create less files, directories

avoid deep project structure

avoid callbacks

Put all logic in controller actions or endpoint handlers

~~Don't~~ Repeat Yourself

Avoid code dependencies

Avoid the layered pyramid

Avoid abstraction and indirection

Put all logic in controller actions or endpoint handlers!

http://yosefk.com/blog/redundancy-vs-dependencies-which-is-worse.html

TDD - Write less tests

Only test user facing endpoints

Avoids brittle tests

Avoids throwaway tests

Avoids simulated tests

http://rangle.io/blog/how-to-fearlessly-iterate-your-rest-apis-through-http-endpoint-testing/

When you only test endpoints and all your logic is in the endpoint handler:

You know exactly the one place to look to fix an endpoint test failure

You don't have to jump around to different files and classes to see all the code

You don't have to follow the data being passed around to see if it's correct

You are testing the entire feature including the database part. No mocking, no simulated test.

When you make a change to fix the issue, no other code is effected since no code depends on the code that is changed

Drawbacks:

You may have to copy paste your fix to other endpoint handlers
But this is mitigated by the fact that you are making changes in just one layer

(Avoid) Caching

Caching masks inherent performance problems
Cache invalidation strategy is a hard problem
Caching layer interaction adds complexity
Caching hides state change
Caching complicates testing

All of this slows you down when debugging and testing when endpoint tests fail

(Avoid) Background processing

Reduces moving parts you have to manage
No queue and jobs to manage, and to simulate in dev.
No uncertainty about failed jobs
Better visibility of job performed since they are inline in your endpoint handlers
Easier testing

Drawback slows down performance of you endpoint
Mitigate by using async fire and forget techniques

Prefer rapid feature\function delivery over form\style

Avoid client side MVC frameworks coupled to Restful APIs

Prefer Server side generated HTML
with a dash of AJAX\PJAX for interactivity

Use jQuery and Bootstrap

Avoiding client side javascript:
Optimizes for speed of iteration, minimizes time to feature release and reduces moving parts

Bonus:
If users use a feature that does not have a modern UI,
then that feature is providing them real value despite the UI.
Provided the UI is easy to use and reliable.

River

Web app framework for continuous delivery

based on Flask web framework and Heroku application hosting platform

open source

https://github.com/aregsar/river