# FINISHING GIT AND MOVING TO USING RAILS TO SUPPORT DESIGN DECISIONS

# BACK TO GIT

# Pull = Fetch + Merge

- Merge two repos = try to apply commits in either one to both
  - Conflict if different changes to same file "too close" together
  - `git pull` == `git pull origin master`
- Successful merge implies commit!
  - Always commit before merging/pulling
  - Commit early & often—small commits OK!
  - READ THE RESPONSES FROM GIT!!!!

*

# Commit: a tree snapshot identified by a commit-ID

➤40-digit hex hash (SHA-1), unique in the universe…but a pain

➤use unique (in this repo) prefix, eg 770dfb

HEAD: most recently committed version on current branch

ORIG_HEAD: right after a merge, points to pre-merged version

HEAD~***n***: n'th previous commit

770dfb~2: 2 commits before 770dfb

"master@{01-Sep-2012}": last commit prior to that date

*

# Undo!

git reset --hard ORIG_HEAD

git reset --hard HEAD

git checkout *commit-id* -- *files…*

➤Comparing/sleuthing:

git diff *commit-id* -- *files…*

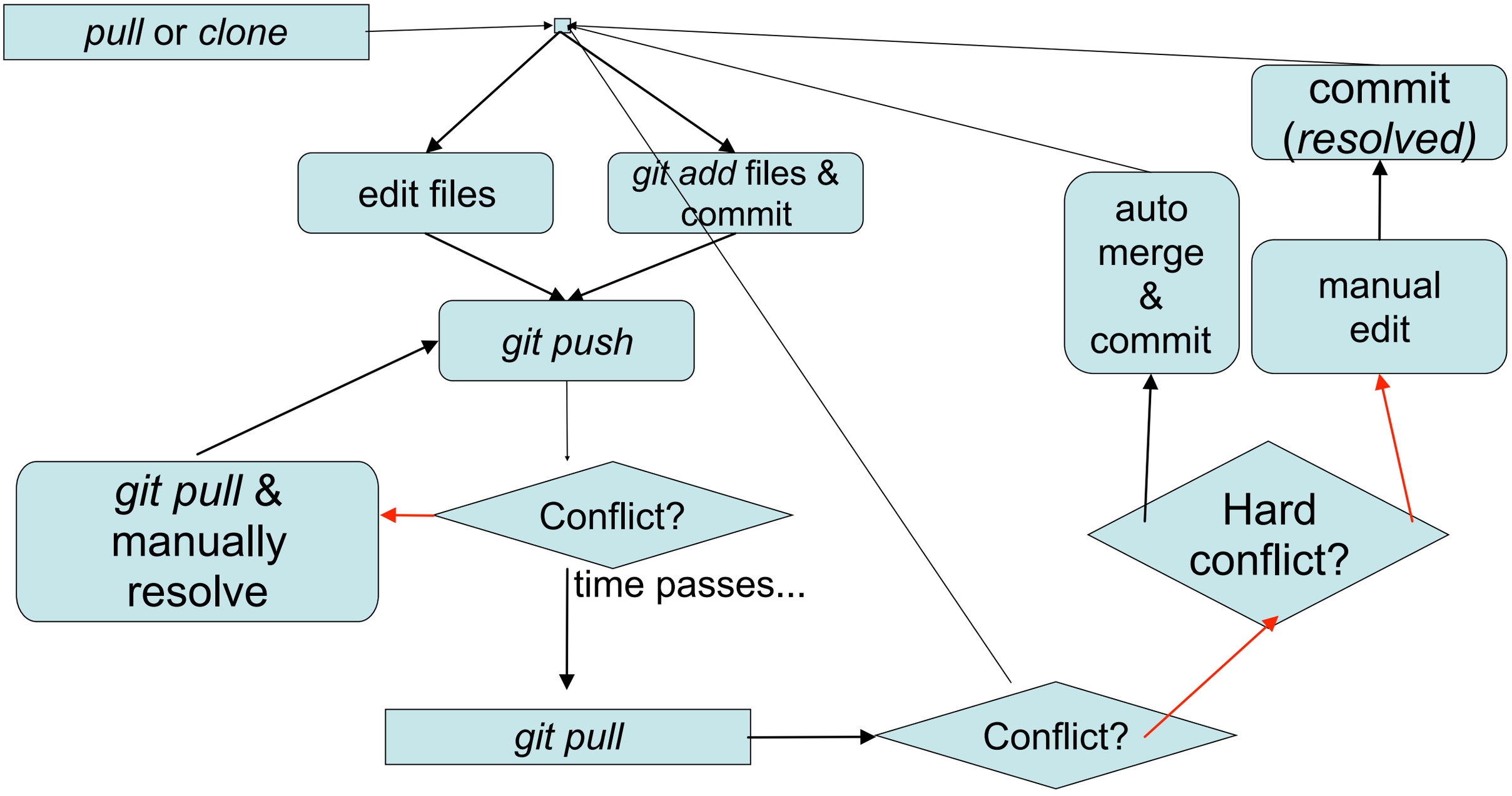
git diff "master@{01-Sep-12}" -- *files*

git blame *files*

git log *files*

*

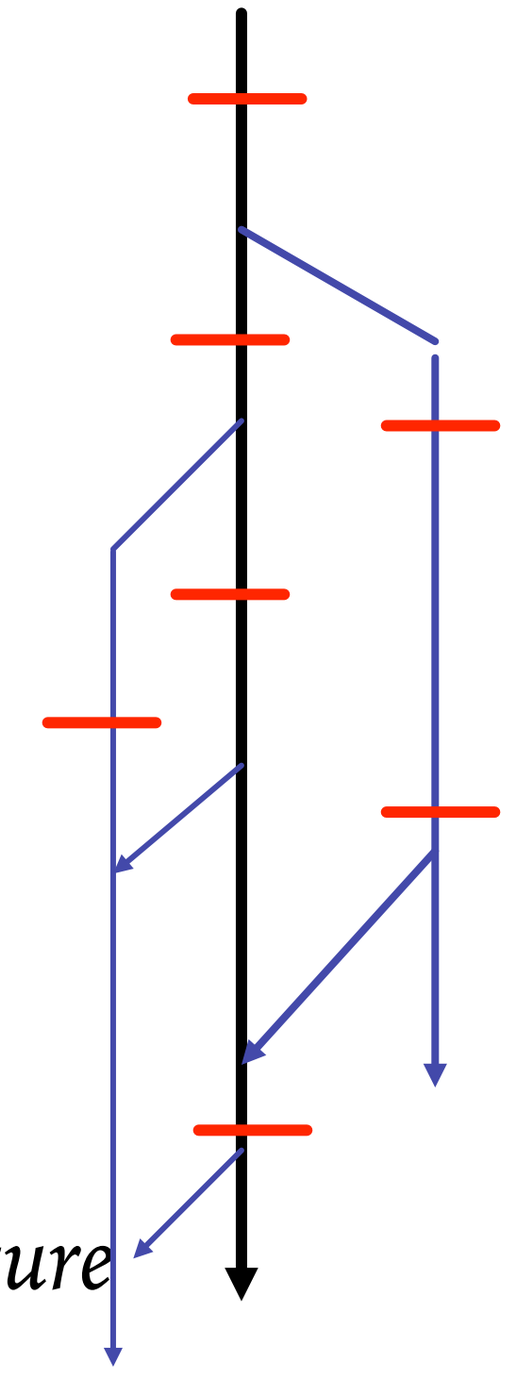
# Version control with conflicts



# Effective Branching

# Branches

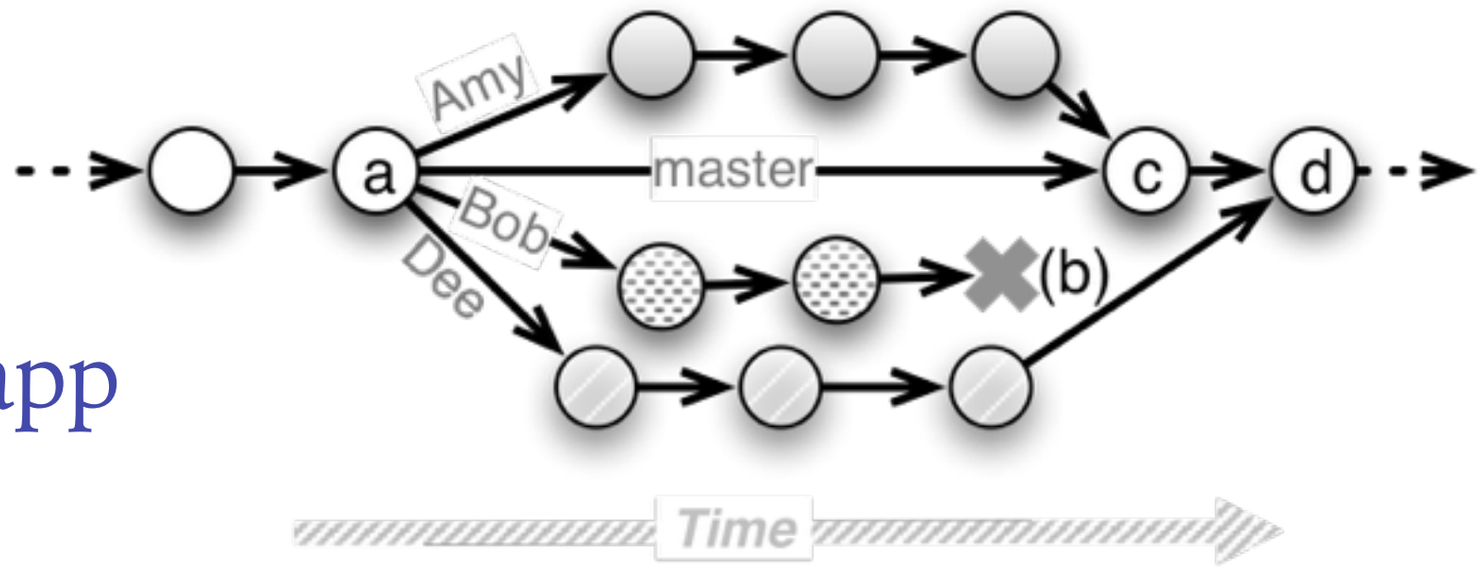
- Development **trunk** vs. **branches**
  - trunk is called “master branch” in Git
  - Creating branch is *cheap!*
  - switch among branches: *checkout*
- Separate commit histories per *branch*
- *Merge* branch back into trunk
  - ...or with *pushing* branch changes
  - Most branches eventually die
- Killer use case for agile SaaS: *branch per feature*



# Creating new features without disrupting working code

1. To work on a new feature, create new branch *just for that feature*
   - ➤many features can be in progress at same time
2. Use branch *only* for changes needed for *this feature*, then merge into trunk
3. Back out this feature ⇔ undo this merge

In well-factored app,
1 feature shouldn't
touch many parts of app



# Mechanics

- Create new branch & switch to it

```
git branch CoolNewFeature
```

```
git checkout CoolNewFeature ←current branch
```

- Edit, add, make commits, etc. on branch
- Push branch to origin repo (optional):

```
git push origin CoolNewFeature
```

  - creates *tracking branch* on remote repo

- Switch back to master, and merge:

```
git checkout master
```

```
git merge CoolNewFeature ← warning!!
```

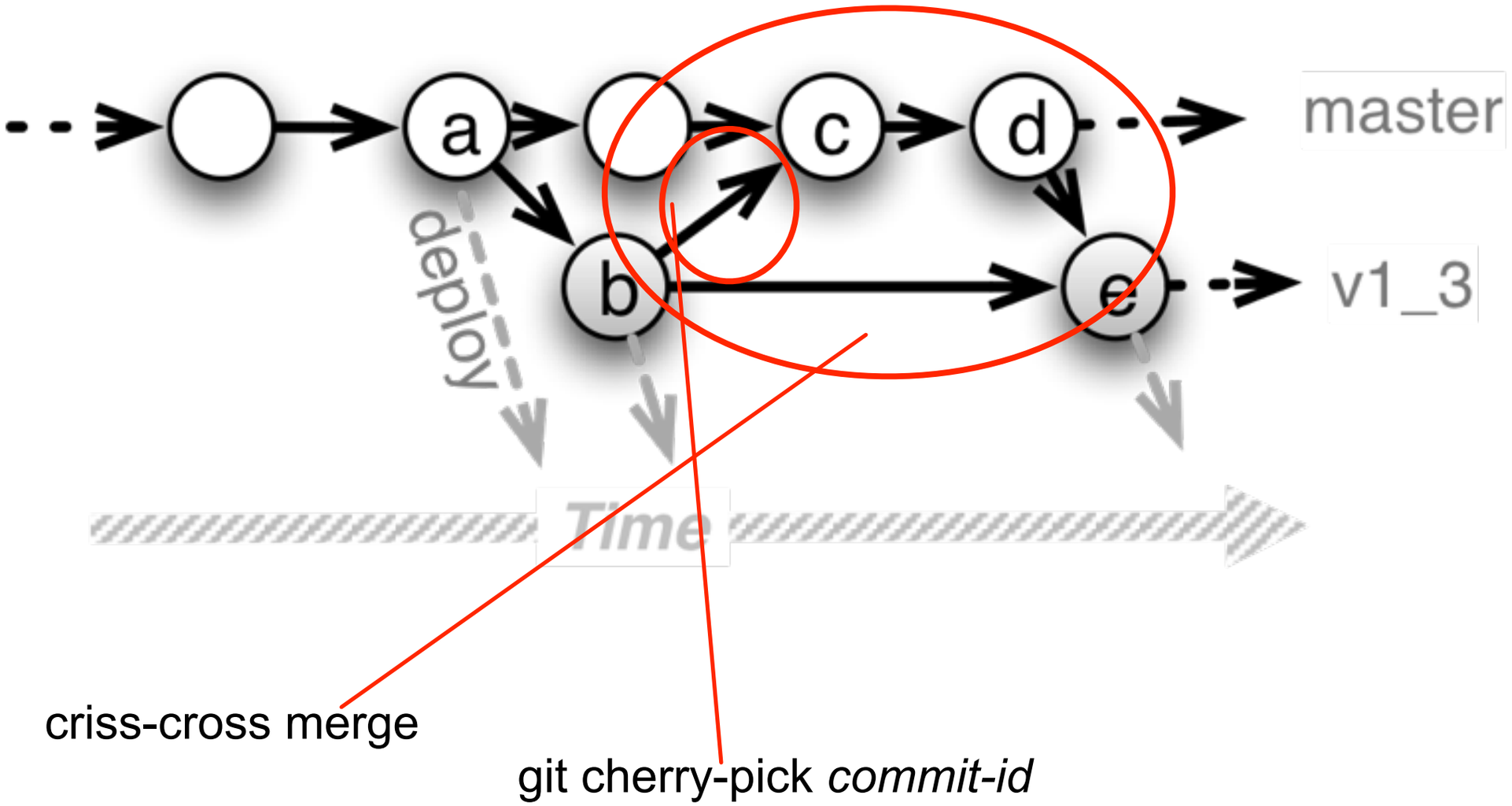
*

# Branches & Deployment

- Feature branches should be short-lived
  - otherwise, drift out of sync with master, and hard to reconcile
  - git rebase can be used to "incrementally" merge
    - git cherry-pick can be used to merge only specific commits
- "Deploy from master" is most common

*

# Release/bugfix branches and cherry-picking commits




Rationale: release branch is a stable place to do incremental bug fixes

*

# Branch vs. Fork

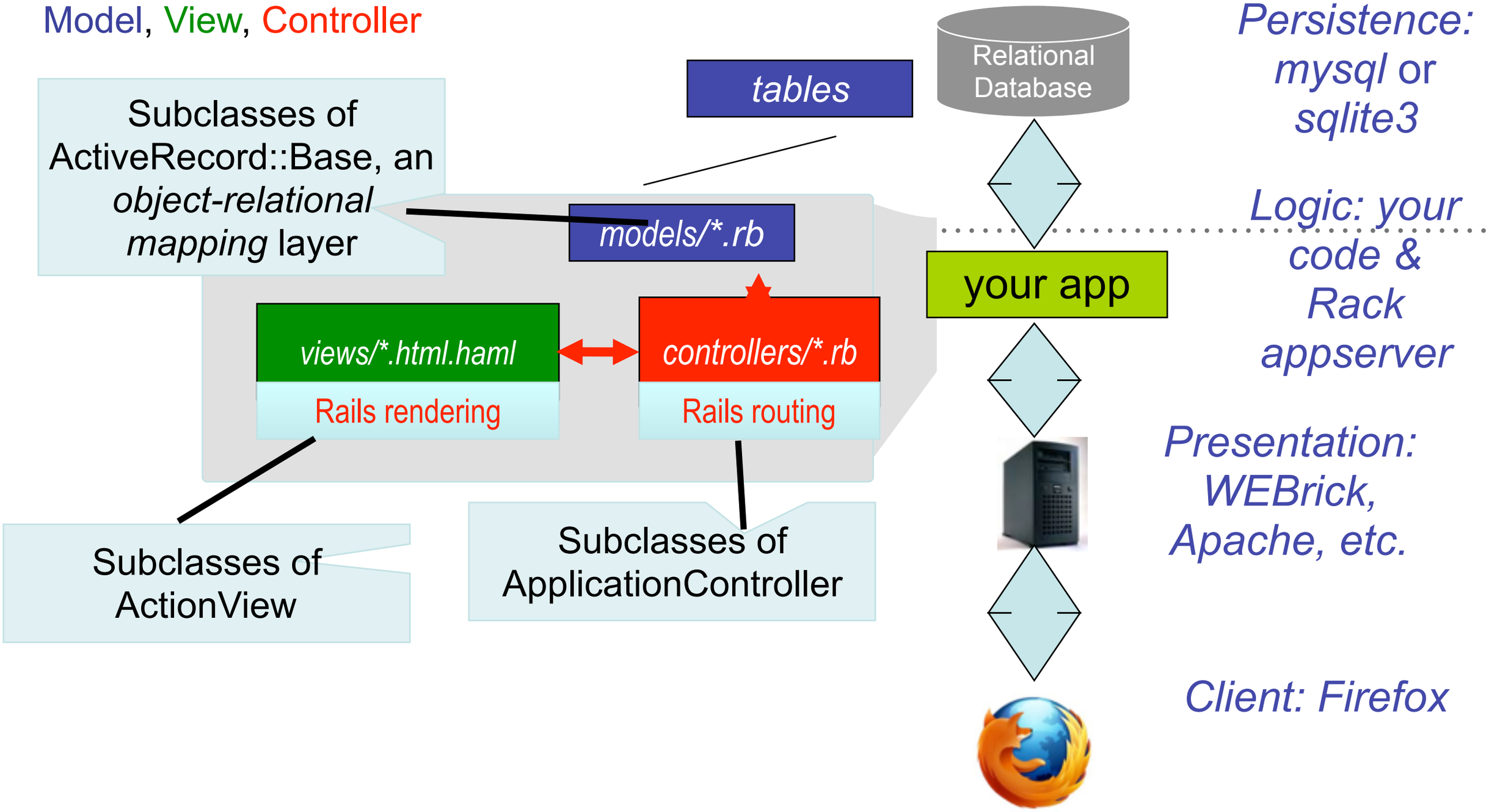
- Git supports *fork & pull* collaboration model
  - branch: create temporary branch in *this repo*
  - merge: fold branch changes into master (or into another branch)
  - fork: clone *entire repo*
  - pull request: I ask you to pull specific commits from my forked repo

*

# Rails from Zero to CRUD

# Rails as an MVC Framework

Model, View, Controller




*Persistence: mysql* or *sqlite3*

*Logic: your code & Rack appserver*

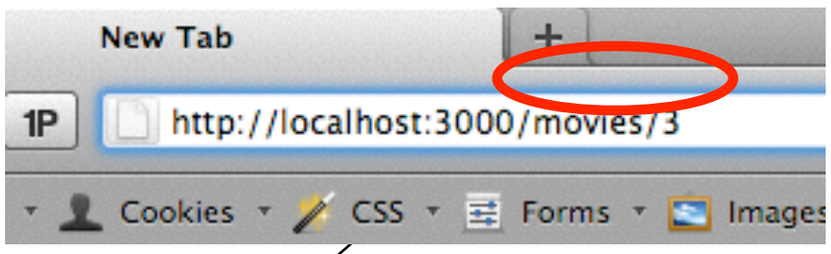
*Presentation: WEBrick, Apache, etc.*

*Client: Firefox*

# A trip through a Rails app

1. Routes (in routes.rb) map incoming URL's to *controller actions* and extract any optional *parameters*
   - Route's "wildcard" parameters (eg :id), plus any stuff after "?" in URL, are put into params[] hash accessible in controller actions
2. Controller actions set *instance variables,* visible to *views*
   1. Subdirs and filenames of views/ match controllers & action names

- Controller action eventually *renders* a view




config/routes.rb

```
GET /movies/:id
{:action=>'show',:controller=>'movies'}
```

app/controllers/movies_controller.rb

```
def show id = params[:id]
 @mv=Movie.find(id)
end
```

app/views/movies/show.html.haml

```
%li
 Rating:
 = @mv.rating
```

# Databases & Migrations

## The Database is Golden

➤Contains valuable customer data—don't want to test your app on that!

➤Rails solution: development, production and test *environments* each have own DB

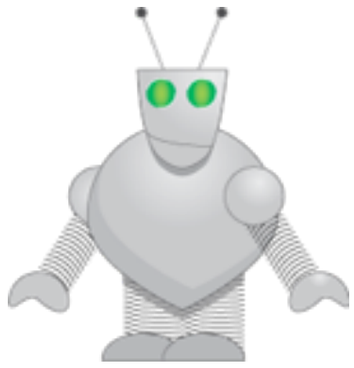
➤Different DB types appropriate for each

➤How to make *changes* to DB, since will have to repeat changes on production DB?

➤Rails solution: *migration*—script describing changes, portable across DB types

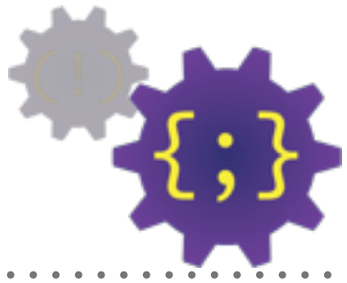
# Migration Advantages

- Can identify each migration, and know which one(s) applied and when
  - Many migrations can be created to be *reversible*
- Can manage with version control
- *Automated == reliably repeatable*
  - Compare: use Bundler vs. manually install libraries/gems



- Theme: *don't do it—automate it*
  - *specify* what to do, create tools to automate

# Meet a Code Generator



## rails generate migration CreateMovies

➤Note, this just *creates* the migration. We haven't *applied* it.

http://pastebin.com/VYwbc5fq

➤Apply migration to development:**rake db:migrate**

➤ Apply migration to production:**heroku rake db:migrate**

➤Applying migration also records in DB itself which migrations have been applied

# EXAMPLE

```
class CreateMovies < ActiveRecord::Migration
  def up
    create_table 'movies' do |t|
      t.string 'title'
      t.string 'rating'
      t.text 'description'
      t.datetime 'release_date'
      # Add fields that let Rails automatically keep track
      # of when movies are added or modified:
      t.timestamps
    end
  end
  def down
    drop_table 'movies' # deletes the whole table and all its data!
  end
end
```

# Rails Cookery #1

➤ Augmenting app functionality ==
adding models, views, controller actions

To *add a new model* to a Rails app:

(or change/add attributes of an existing model)

1. Create a migration describing the changes:

`rails generate migration` (gives you boilerplate)

2. Apply the migration: `rake db:migrate`

3. If new model, create model file `app/models/`*model*`.rb`

➤ Update test DB schema: `rake db:test:prepare`