UNIVERSITY OF CALIFORNIA, SAN DIEGO

Operating System monitoring for programming tutorial creation and evaluation

A dissertation submitted in partial satisfaction of the
requirements for the degree
Masters

in

Computer Science

by

Alok Shankar Mysore

Committee in charge:

Professor Philip J. Guo, Chair
Professor Scott R. Klemmer
Professor Leo Porter

2018

The dissertation of Alok Shankar Mysore is approved, and it is acceptable in quality and form for publication on microfilm:

____

____

____
Chair

University of California, San Diego

2018

DEDICATION

lol idk

ACKNOWLEDGEMENTS

People to thank: 1. Philip 2. Scott

VITA

2002	B. S. in Mathematics *cum laude*, University of Southern North Dakota, Hoople
2002-2007	Graduate Teaching Assistant, University of California, San Diego
2007	Ph. D. in Mathematics, University of California, San Diego

PUBLICATIONS

Your Name, “A Simple Proof Of The Riemann Hypothesis”, *Annals of Math*, 314, 2007.

Your Name, Euclid, “There Are Lots Of Prime Numbers”, *Journal of Primes*, 1, 300 B.C.

ABSTRACT OF THE DISSERTATION

Operating System monitoring for programming tutorial creation and evaluation

by

Alok Shankar Mysore
Masters in Computer Science

University of California San Diego, 2018

Professor Philip J. Guo, Chair

This dissertation will be abstract.

Chapter 1

Introduction

This is only a test.

1.1 A section

Lorem ipsum dolor sit amet, consectetuer adipiscing elit. Nulla odio sem, bibendum ut, aliquam ac, facilisis id, tellus. Nam posuere pede sit amet ipsum. Etiam dolor. In sodales eros quis pede. Quisque sed nulla et ligula vulputate lacinia. In venenatis, ligula id semper feugiat, ligula odio adipiscing libero, eget mollis nunc erat id orci. Nullam ante dolor, rutrum eget, vestibulum euismod, pulvinar at, nibh. In sapien. Quisque ut arcu. Suspendisse potenti. Cras consequat cursus nulla.

1.1.1 More Stuff

Blah

Chapter 2

Torta - Generating programming tutorials by OS monitoring

2.1 Formative Interviews and Design Goals

To discover the challenges faced by people who manually create mixed-media software tutorials, we interviewed four teaching assistants responsible for creating web programming tutorials for an introductory HCI course [5].

In this course, students build full-stack web applications with a mix of tools such as Git for version control, Node.js for server-side development, Handlebars for templating, Bootstrap for responsive CSS, and Heroku for live deployment to the web. Students come into the course from a diverse variety of majors and widely varying levels of prior programming experience, so the staff teaches weekly programming tutorials to get everyone acquainted with the mechanics of web development (e.g., setup, coding, testing, online deployment). Since these tutorials must coordinate across multiple command-line and GUI applications, they are representative of the sorts of tutorials that we would like Torta to automatically generate.

Each lesson is a webpage containing a mixed-media tutorial interspersing Power-

Point slides, video clips, and command/code snippets. Everything is created manually: The staff first makes a PowerPoint deck (usually around 100 slides) containing step-by-step instructions, commands to run, code to write/modify, and screenshots showing expected visual outputs. They export the slides as images to embed within a webpage and then supplement it with screencast videos and commands/code that students should copy-and-paste into their terminals. From our interviews with teaching assistants, we found three main bottlenecks in creating these tutorials and deploying them during in-class lab sessions:

PowerPoint slides versus screencast videos: Students greatly preferred reading the PowerPoint slides since those were easily skimmable, but the staff found them far more tedious to create since they needed to first demonstrate their actions and then manually copy-and-paste all commands, code, expected outputs, and screenshots into the slides. Also, sometimes the slides did not go into enough detail or skipped steps due to staff oversight or simply lack of prep time. In contrast, screencast videos were much easier for staff to record and contain all necessary details, but were harder for students to browse. The staff struck a compromise by placing slides and videos side by side on the webpage, using videos to showcase dynamic events such as GUI interactions and web animations.

Slides, videos, and code are disconnected: Besides slides and videos, the staff also embedded snippets of code and commands into tutorial webpages. They did this because students found it hard to copy-and-paste directly from PowerPoint slides due to syntax-breaking formatting issues (e.g., bad line breaks, smart quotes, special characters, incomplete code due to lack of space on slides), and it is impossible to copy from videos. Additionally, the staff maintained a GitHub repository of skeleton starter code and helper scripts for students to build upon when following these tutorials. This heterogeneous setup meant that when the staff created or updated each tutorial, they had to manually keep four disparate data sources all updated and in sync: PowerPoint slides, videos, command/code snippets, and the GitHub repository of skeleton code; these disconnects led to numerous bugs in tutorials.

Hard for students to validate progress: When working through tutorials in class, students were anxious about whether they were following certain steps properly since the PowerPoint slides did not always specify the expected effects of each step, and videos were not available for all steps. Many effects were not immediately visible on-screen, such as the results of running a Heroku configuration command. Even worse, when a student does not follow a step properly, everything may still appear to work, but subtle errors silently propagate and manifest in later steps with unrelated error messages. These problems arise because students cannot easily check their progress. The staff ended up dealing with this by writing *validation scripts* for each tutorial. When a student runs a validation script, it checks that their filesystem state, environment variables, and current directory are what the tutorial expects; otherwise it prints a targeted error message.

These bottlenecks inspired a set of design goals for Torta:

- **D1**: Creating a tutorial should be as easy as recording a screencast video, but tutorials should offer advantages of text-based formats like easy skimming and copy-paste.
- **D2**: A tutorial should automatically encapsulate videos, textual exposition, code examples, and terminal commands together into one package instead of in disconnected silos.
- **D3**: Users should be able to view the tutorial at varying levels of detail to accommodate their own expertise level.
- **D4**: Users should be able to incrementally validate their progress as they follow each step of the tutorial.

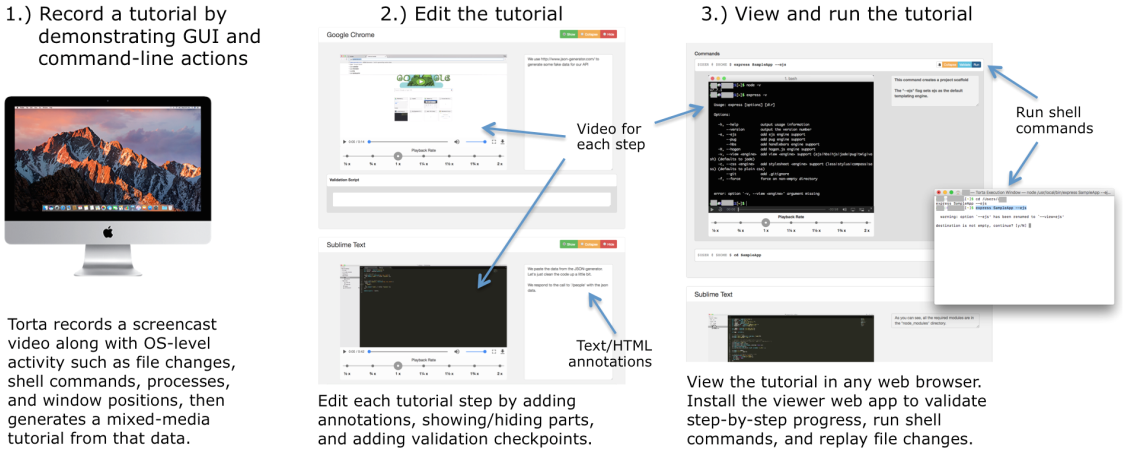

1.) Record a tutorial by demonstrating GUI and command-line actions
2.) Edit the tutorial
3.) View and run the tutorial
Video for each step
Run shell commands
Text/HTML annotations
Torta records a screencast video along with OS-level activity such as file changes, shell commands, processes, and window positions, then generates a mixed-media tutorial from that data.
Edit each tutorial step by adding annotations, showing/hiding parts, and adding validation checkpoints.
View the tutorial in any web browser. Install the viewer web app to validate step-by-step progress, run shell commands, and replay file changes.

Figure 2.1: Torta allows macOS users to create mixed-media tutorials by demonstration, and then edit, view, and run those tutorials in a web browser.

2.2 The Design and Implementation of Torta

Torta consists of three components: a tutorial recorder, editor, and viewer (Figure 2.1). We now describe each in turn.

2.2.1 Tutorial Recorder

Torta's recorder allows the user to record a tutorial just as easily as recording a screencast video (Design Goal D1). Our prototype is implemented for macOS using AppleScript, Python, Bash, and DTrace [2] scripts to perform OS-level activity tracing. It should be straightforward to port this OS-wide tracing-based approach to other operating systems.

When the user wants to start recording a tutorial, they activate Torta by running a terminal command, which immediately launches a set of activity tracers. The user then records their tutorial by simply demonstrating actions on their computer, and the tracers log the following data in the background:

- **Screencast video recorder**: Torta uses Apple's built-in Quicktime app to record a standard full-screen screencast video with audio narration and mouse clicks visualized.

- **Foreground GUI window monitor**: The position and dimensions of the user's current foreground GUI window are logged once per second, along with the process ID of the program that owns the current foreground window.
- **Keystroke logger**: All user keystrokes are logged.
- **Shell command logger**: The contents of all terminal commands run in any shell are logged and timestamped. The current working directory, username, and environment variables used for running each command are also logged. Our current logger works for Bash (the default on macOS) and Zsh, but can be easily extended to other custom shells.
- **Filesystem activity tracer**: Torta uses DTrace [2] to record a subset of system calls that access the filesystem. Specifically, it logs the timestamps, owner process IDs, and parameters of the following filesystem-related system calls: `open()`, `write()`, `close()`, `rename()`, and `unlink()` (for opening, writing to, closing, renaming, and deleting files, respectively). Torta makes a timestamped backup copy of each affected file after the respective system call is run. This feature is useful for saving all versions of files that users edit within interactive applications such as text editors, IDEs, or Photoshop: Each time the user presses "Save" within the app, a `write()` system call occurs, and Torta saves a backup copy, which lets it later display diffs.
- **OS process tree logger**: Torta logs the command names, start/end timestamps, process IDs (PIDs), and parent process IDs (PPIDs) of all OS processes launched after the user activates Torta. This log serves two purposes: First, it filters the system call trace (see above) to consider only processes that launched *after* the user activated Torta, which eliminates the noise from dozens of irrelevant system-wide processes previously running on the user's machine. Second,

> it is necessary for linking the system call trace to foreground GUI windows. Here is why: Many interactive apps adopt a multi-process model for robustness. For instance, Google Chrome launches one OS process per browser tab, and text editors such as Sublime Text launch one OS process per text editor tab along with a separate process for the GUI. Thus, the process that owns the Sublime Text foreground GUI window is *not* the process that makes the `write()` system calls to save the user's files. Torta can use the OS process tree of PIDs and PPIDs to link Sublime Text's user-initiated file save events with its GUI window, since they are owned by sibling processes.

After the user finishes recording their demonstration and shuts down Torta, it automatically creates a *mixed-media tutorial* by post-processing and combining the recorded data into self-contained package that contains all traces, segmented videos, and saved file versions (Design Goal D2). As shown in Figure 2.2, a Torta-generated tutorial has a hierarchical structure that aims to follow the design guidelines of Chi et al. [3]:

Top-level steps – foreground GUI windows: A Torta tutorial is an ordered list of top-level steps. Each step spans the duration of one foreground GUI window. Torta uses FFmpeg [1] to split the screencast video into one mini-video per foreground window duration and crops those videos to show only the foreground window. We felt that foreground windows were the most natural step boundaries for these kinds of software tutorials, since users often perform a set of actions within one window (e.g., an IDE) and then switch to another window (e.g., Photoshop) to perform the next set.

Each step is rendered as a mini-video along with a *filesystem tree* showing which files were added, deleted, renamed, and modified by processes associated with the foreground GUI window during that step (Figure 2.4). We chose to visualize filesystem changes since those represent the persistent effects of user actions within an application. Regardless of what kind of app the user is running, if some action has

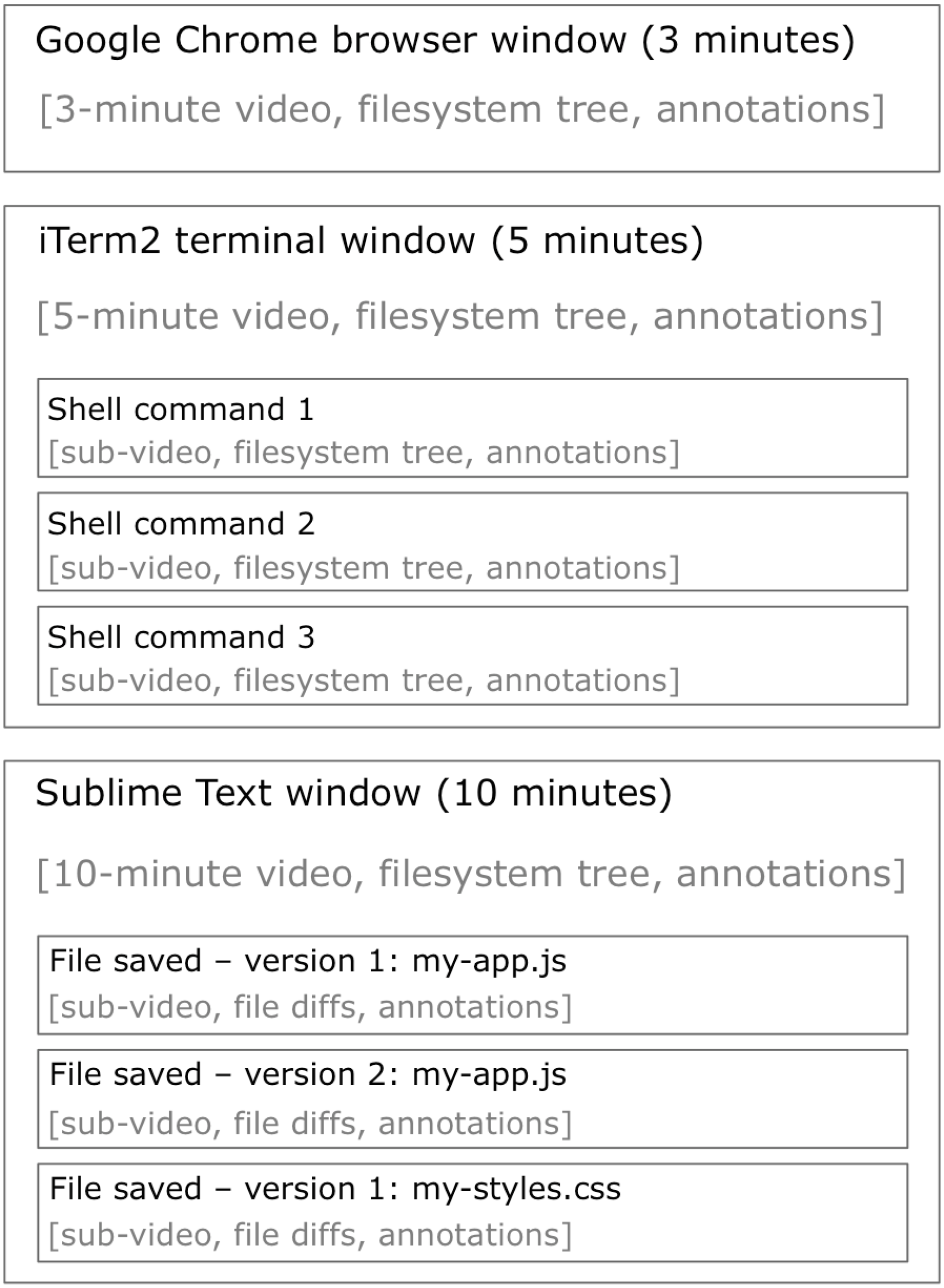
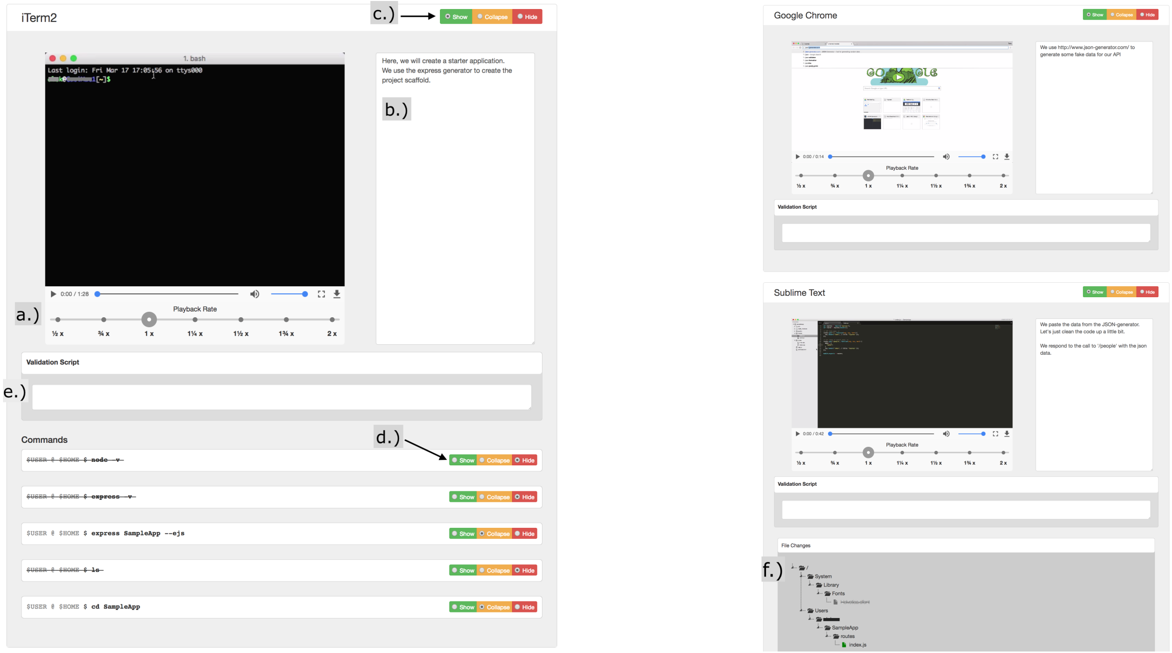

Google Chrome browser window (3 minutes)
[3-minute video, filesystem tree, annotations]
iTerm2 terminal window (5 minutes)
[5-minute video, filesystem tree, annotations]
Shell command 1
[sub-video, filesystem tree, annotations]
Shell command 2
[sub-video, filesystem tree, annotations]
Shell command 3
[sub-video, filesystem tree, annotations]
Sublime Text window (10 minutes)
[10-minute video, filesystem tree, annotations]
File saved – version 1: my-app.js
[sub-video, file diffs, annotations]
File saved – version 2: my-app.js
[sub-video, file diffs, annotations]
File saved – version 1: my-styles.css
[sub-video, file diffs, annotations]

Figure 2.2: Example structure of a mixed-media tutorial generated by Torta. Each of the three steps represents a foreground GUI window duration. There are three sub-steps within the terminal and IDE windows.



```
Last login: Fri Mar 17 17:05:56 on ttys000
[redacted]@[redacted][~]$
```

Here, we will create a starter application. We use the express generator to create the project scaffold.

~~$USER @ $HOME $ node -v~~

~~$USER @ $HOME $ express -v~~

$USER @ $HOME $ express SampleApp --ejs

~~$USER @ $HOME $ ls~~

$USER @ $HOME $ cd SampleApp

We use http://www.json-generator.com/ to generate some fake data for our API

We paste the data from the JSON-generator. Let's just clean the code up a little bit.

We respond to the call to '/people' with the json data.

```
/
├── System
│   └── Library
│       └── Fonts
│           └── ~~Helvetica.dfont~~
└── Users
    └── [redacted]
        └── SampleApp
            └── routes
                └── index.js
```

Figure 2.3: Zoomed-in screenshots of Torta's tutorial editor showing three steps (i.e., foreground windows): iTerm2, Google Chrome, and Sublime Text. Each step contains: a.) Video player with playback speed adjuster, b.) Text annotation box, c.) Toggle to show/collapse/hide this entire step, d.) Toggle to show/collapse/hide each sub-step (here the hidden shell command sub-steps are crossed-out), e.) Validation script, f.) Filesystem tree (see Figure 2.4).

a lasting effect on their computer, it will likely manifest in the filesystem.

Sub-steps: Torta further splits each top-level step into sub-steps based on two common kinds of user actions (Figure 2.2):

- ***Shell commands***: If the user runs multiple shell commands within the duration of one foreground window (usually some kind of terminal app), Torta splits that step into one sub-step for each command. Each sub-step is shown as a mini-video spanning the duration of only that command, the text of that command, its current working directory, environment variables, and a filesystem tree showing what files that command added, deleted, renamed, and modified.
- ***File saves***: When the foreground window is an interactive app such as an IDE, web browser, or image editor, the user may be editing files and periodically saving their progress to disk. Torta splits each step into sub-steps based on file save events, treating saves like user-defined checkpoints in the tutorial. Again, each sub-step gets its own mini-video. If the saved file is plain text, Torta also shows the diffs between the current and previously-saved versions, which is useful for showing edits in code and configuration files.

2.2.2 Tutorial Editor

The mixed-media tutorial that Torta automatically generates from the user's demonstration is already complete and ready to view on the web. One can think of it as a screencast video that is segmented and enhanced with OS-level trace data. However, it can be hard for users to record a pristine, error-free video in one take. Furthermore, users also want to augment tutorials with textual annotations and other customizations. To fulfill these needs, Torta provides a tutorial editor, which renders the tutorial just as the viewer would see it but adds extra controls for the following actions (Figure 2.3):

Adding text annotations to steps/sub-steps: The user should already provide audio narration when recording their demonstration, which will show up in the screencast video. The editor also lets them add Markdown-based rich-text annotations next to the segmented video for each step/sub-step.

Hiding steps/sub-steps: The user can hide any step/sub-step from the viewer to eliminate mistakes or redundancies (effectively deleting them from the edited tutorial). If the user hides a step that is in between two steps that belong to the same application, then those two surrounding steps get merged into one. This happens when, say, the user is in an IDE, then switches to a web browser to look up something quickly, then switches back to the IDE. If the user hides the web browser step because they deem it irrelevant for the tutorial, then the two IDE steps get merged together as one step in the viewer. Torta does not support post-hoc re-recording of steps in the editor. A workaround is to record an entire session even with errors included and then hide erroneous steps using the editor.

Collapsing steps/sub-steps: If the user deems certain steps or sub-steps to be less important for the tutorial, they can show them in a collapsed form. The viewer will see those steps as a collapsed summary but can manually un-collapse them to dive into details. Torta displays compact summaries so that viewers can more easily skim step contents (e.g., “Photoshop window active for 2 minutes, modified 3 files").

Torta implements heuristics to automatically collapse certain steps/sub-steps that are likely to be less important to the tutorial. For instance, if a shell command does not make any changes to the filesystem (e.g., `ls` or `git status`), it is collapsed by default since the user was probably checking their setup before proceeding to the next step. Also, if any GUI window was in the foreground for less than 5 seconds, had less than 10 user keystrokes, and did not modify the filesystem, then its step is also collapsed by default. This filters out “flickers" where the user switches between windows momentarily to quickly check something before the next step.

Collapsing filesystem tree components: Recall that Torta displays a filesys-

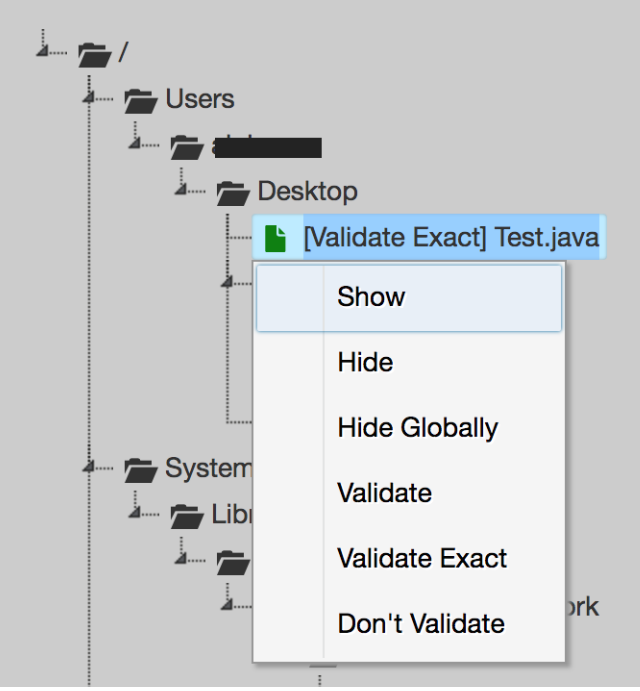


Figure 2.4: Each file-modifying tutorial step displays a filesystem tree of all files affected by running that step. In the editor UI, the user can right-click to show/hide files and to mark for validation.

tem tree within each step and sub-step that modifies the filesystem (Figure 2.4). However, during pilot testing we noticed that some commands (e.g., `git clone`) can affect hundreds of files, so their trees are extremely large. To reduce visual overload, the user can collapse tree nodes to hide and summarize their sub-trees. For instance, the user can collapse a `.git/` sub-directory to see a summary like “100 files added and 15 files modified in `.git/`." Just as with collapsed steps, the viewer can un-collapse tree nodes to see more details on demand. The user can also choose “Hide Globally" to hide a particular file/directory across all tutorial steps.

Adding validation: The editor provides two ways to specify how people (i.e., tutorial *consumers*) can validate progress at each step as they are following the tutorial (Design Goal D4):

1. Marking files to validate: The user can mark each file in the filesystem tree of a step/sub-step as “Validate", “Validate Exact," or “Don't Validate" (Figure 2.4). If the user marks a directory, everything within it also gets marked with that label. “Validate" means that Torta should check that the consumer's file gets altered in the way that this step specifies (e.g., modified or renamed), and “Validate Exact" means that the new contents of the file should also exactly match the saved version bundled in the tutorial package. For example, in a step where the consumer is supposed to add their username to a section within a configuration file, that file should be marked as “Validate" to check that it has been modified, but not “Validate Exact" since everyone's username will be different.

2. Writing validation scripts: File-based validation handles the most common uses, but if tutorial creators want more flexibility, they can write a validation script for each step/sub-step (Figure 2.3e). This is a Bash script that will run on the consumer's machine to check that their OS state is as expected.

This feature is similar in spirit to the step-level validation features offered by tutorial systems for other domains [4, 6].

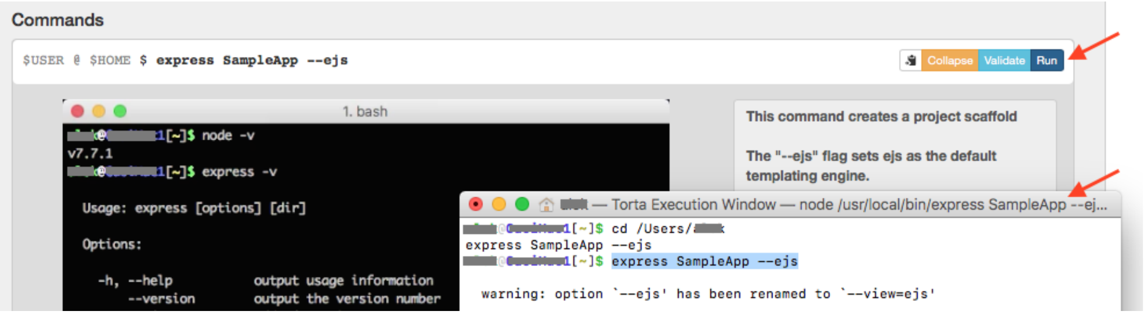
After the user finishes editing the tutorial, they can publish it as a webpage or send the self-contained package to viewers.

2.2.3 Tutorial Viewer

Since Torta tutorials are ordinary webpages, they can be viewed in any browser. Each tutorial initially loads with certain steps/sub-steps collapsed, certain file tree nodes collapsed, and each video playing at the speed pre-set by the creator. However, the user can adjust any of those settings. In addition, they can click on any file in the tutorial and view/download the version of that file present during that respective step (all versions are stored in the package). This ability to selectively hide and show details was inspired by a challenge discovered during formative interviews: Students preferred seeing varying levels of detail depending on their expertise level (Design Goal D3). It is hard to achieve this flexibility with raw screencast videos or PowerPoint slides.

To make tutorials more readable, Torta canonicalizes all file paths within command invocations and filesystem trees. For instance, when Alice creates a tutorial, many of her file paths will contain `/home/alice` if they are within her home directory. But when Bob is viewing the tutorial, he would prefer to see paths starting with `/home/bob` instead of `/home/alice`. Torta canonicalizes paths by replacing the creator's home directory with the `$HOME` variable. Additionally, the creator can use the tutorial editor to specify other path variables to replace. One use case is specifying a `$PROJECT_ROOT` directory where all files within a project should live. The tutorial viewer prompts the user to enter their own preferred values for all of these variables and rewrites all paths within the webpage accordingly. Note that `$HOME` and other environment variables are automatically set if the tutorial is loaded from the user's machine rather than viewed on the web.

If the user downloads the tutorial to their macOS machine and loads it via the Torta viewer web app on localhost, then they can access two additional features as


Collapse
Validate
Run
1. bash
[redacted] — Torta Execution Window — node /usr/local/bin/express SampleApp --ej...

Commands

```
$USER @ $HOME $ express SampleApp --ejs
```

```
[redacted]@[redacted]1[~]$ node -v
v7.7.1
[redacted]@[redacted]1[~]$ express -v

  Usage: express [options] [dir]

  Options:

    -h, --help          output usage information
        --version       output the version number
```

This command creates a project scaffold

The "--ejs" flag sets ejs as the default templating engine.

```
[redacted]@[redacted]1[~]$ cd /Users/[redacted]
express SampleApp --ejs
[redacted]@[redacted]1[~]$ express SampleApp --ejs

  warning: option `--ejs' has been renamed to `--view=ejs'
```

Figure 2.5: Each step and sub-step contains “Validate" and “Run" buttons on the upper right. Here when the user clicks on “Run" for a shell command sub-step, Torta runs that commands in a new terminal window.

shown in Figure 2.5:

1. Validating step-by-step progress: After manually performing the actions specified by a particular step/sub-step, the user can click the “validate" button alongside its video. Torta will check that the affected files on the user’s local filesystem have been modified in the ways that the creator originally expected (i.e., specified via “validate" and “validate exact" labels in the filesystem tree) and also run the validation script if it exists. Then it prompts the user if there are errors and offers to overwrite any mismatched files with the versions from the tutorial package if the user wishes. This capability lets the user check that they are properly following along with each step of the tutorial and to catch bugs earlier (Design Goal D4).

2. Automatically running steps: The user can click the “run" button next to each step/sub-step to have Torta automatically run that step for them. For a shell command, Torta launches a terminal app on the user’s machine and runs the command from that terminal after setting the proper working directory and environment variables. For a step involving a GUI application, Torta does not try to replay GUI actions but rather simply mutates the user’s filesystem in the way that has been prescribed by that step. Although this approach is not always guaranteed to be fully faithful to that step’s actions, in practice it works well in some cases since the persistent effects of a GUI application usually manifest in the filesystem. For instance, if someone demonstrates how to use a GUI to customize the configuration of a complex interactive application, the effects of that customization may show up as changes to some config file. When the user hits “run" on that step, Torta simply copies over the updated version of that config file.

Bibliography

[1] 2017. FFmpeg: A complete, cross-platform solution to record, convert and stream audio and video. `https://ffmpeg.org/`. (2017).

[2] Bryan M. Cantrill, Michael W. Shapiro, and Adam H. Leventhal. 2004. Dynamic Instrumentation of Production Systems. In *Proceedings of the Annual Conference on USENIX Annual Technical Conference (ATEC '04)*. USENIX Association, Berkeley, CA, USA. `http://dl.acm.org/citation.cfm?id=1247415.1247417`

[3] Pei-Yu Chi, Sally Ahn, Amanda Ren, Mira Dontcheva, Wilmot Li, and Björn Hartmann. 2012. MixT: Automatic Generation of Step-by-step Mixed Media Tutorials. In *Proceedings of the 25th Annual ACM Symposium on User Interface Software and Technology (UIST '12)*. ACM, New York, NY, USA, 93–102. `DOI:http://dx.doi.org/10.1145/2380116.2380130`

[4] Jennifer Fernquist, Tovi Grossman, and George Fitzmaurice. 2011. Sketch-sketch Revolution: An Engaging Tutorial System for Guided Sketching and Application Learning. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 373–382. `DOI:http://dx.doi.org/10.1145/2047196.2047245`

[5] Scott Klemmer. 2017. UCSD Interaction Design COGS120/CSE170 - Winter 2017. `http://ixd.ucsd.edu/home/w17/index.php`. (2017).

[6] Suporn Pongnumkul, Mira Dontcheva, Wilmot Li, Jue Wang, Lubomir Bourdev, Shai Avidan, and Michael F. Cohen. 2011. Pause-and-play: Automatically Linking Screencast Video Tutorials with Applications. In *Proceedings of the 24th Annual ACM Symposium on User Interface Software and Technology (UIST '11)*. ACM, New York, NY, USA, 135–144. `DOI:http://dx.doi.org/10.1145/2047196.2047213`