Open edX Developer's Guide

unknown

Dec 09, 2022

CONTENTS

CHAPTER

ONE

GENERAL INFORMATION

1.1 Read Me

The *edX Developer Documentation* is created using RST files and Sphinx. You, the user community, can help update and revise this documentation project on GitHub.

https://github.com/openedx/edx-documentation/tree/master/en_us/developers/source

The edX documentation team welcomes contributions from Open edX community members. You can find guidelines for how to contribute to edX Documentation in the GitHub edx/edx-documentation repository.

1.2 Other edX Resources

Learners, course teams, researchers, developers: the edX community includes groups with a range of reasons for using the platform and objectives to accomplish. To help members of each group learn about what edX offers, reach goals, and solve problems, edX provides a variety of information resources.

To help you find what you need, browse the edX offerings in the following categories.

All members of the edX community are encouraged to make use of the resources described in this preface. We welcome your feedback on these edX information resources. Contact the edX documentation team at docs@edx.org.

1.2.1 Resources for edx.org Learners

Documentation

The edX Help Center for Learners includes topics to help you understand how to use the edX learning management system. The Help Center is also available when you select **Help** while you are in a course, and from your edX dashboard.

In a Course

If you have a question about something you encounter in an edX course, try these options for getting an answer.

Note: If you find an error or mistake in a course, contact the course staff by adding a post in the course discussions.

- Check the **Course** page in the course. Course teams use this page to post updates about the course, which can include explanations about course content, reminders about when graded assignments are due, or announcements for upcoming events or milestones.
- Look for an "Introduction", "Overview", or "Welcome" section in the course content. In the first section in the course, course teams often include general information about how the course works and what you can expect, and also what they expect from you, in the first section in the course.
- Participate in the course discussions. Other learners might be able to answer your question, or might have the same question themselves. If you encounter an unfamiliar word, phrase, or abbreviation, such as "finger exercise" or "board work", search for it on the **Discussion** page, or post a question about it yourself. Your comments and questions give the course team useful feedback for improving the course.
- Investigate other resources. Some courses have a wiki, which can be a good source of information. Outside of the course, a course-specific Facebook page or Twitter feed might be available for learners to share information.

Resources on the edx.org Website

To help you get started with the edX learning experience, edX offers a course (of course!). You can find the edX Demo course on the edx.org website.

When you are working in an edX course, you can select **Help** to access a help center with frequently asked questions and answers.

If you still have questions or suggestions, you can contact the edX Support team for help.

For opportunities to meet others who are interested in edX courses, check the edX Global Community meetup group.

1.2.2 The edX Partner Portal

The edX Partner Portal is the destination for partners to learn, connect, and collaborate with one another. Partners can explore rich resources and share success stories and best practices while staying up-to-date with important news and updates.

To use the edX Partner Portal, you must register and request verification as an edX partner. If you are an edX partner and have not used the edX Partner Portal, follow these steps.

1. Visit partners.edx.org, and select **Create New Account**.
2. Select **Request Partner Access**, then fill in your personal details.
3. Select **Create New Account**. You will receive a confirmation email with your account access within 24 hours.

After you create an account, you can sign up to receive email updates about edX releases, news from the product team, and other announcements. For more information, see *Release Announcements by Email*.

Course Team Support in the edX Partner Portal

EdX partner course teams can get technical support in the edX Partner Portal. To access technical support, submit a support ticket, or review any support tickets you have created, go to partners.edx.org and select **Course Staff Support** at the top of the page. This option is available on every page in the Partner Portal.

1.2.3 The Open edX Portal

The Open edX Portal is the destination for learning about hosting an Open edX instance, extending the edX platform, and contributing to Open edX. In addition, the Open edX Portal provides product announcements and other community resources.

All users can view content on the Open edX Portal without creating an account and logging in.

To comment on blog posts or the edX roadmap, or subscribe to email updates, you must create an account and log in. If you do not have an account, follow these steps.

1. Visit open.edx.org/user/register.
2. Fill in your personal details.
3. Select **Create New Account**. You are then logged in to the Open edX Portal.

Release Announcements by Email

To receive and share product and release announcements by email, you can subscribe to announcements on one of the edX portal sites.

1. Create an account on the Open edX Portal or the edX Partner Portal as described above.
2. Select **Community** and then **Announcements**.
3. Under **Subscriptions**, select the different types of announcements that you want to receive through email. You might need to scroll down to see these options.
4. Select **Save**.

You will now receive email messages when new announcements of the types you selected are posted.

1.2.4 System Status

For system-related notifications from the edX operations team, including outages and the status of error reports. On Twitter, you can follow @edxstatus.

Current system status and the uptime percentages for edX servers, along with the Twitter feed, are published on the edX Status web page.

1.2.5 Resources for edx.org Course Teams

Course teams include faculty, instructional designers, course staff, discussion moderators, and others who contribute to the creation and delivery of courses on edx.org or edX Edge.

The edX Course Creator Series

The courses in the edX Course Creator Series provide foundational knowledge about using the edX platform to deliver educational experiences. These courses are available on edx.org.

- *edX101: Overview of Creating a Course*
- *StudioX: Creating a Course with edX Studio*
- *BlendedX: Blended Learning with edX*
- *VideoX: Creating Video for the edX Platform*

edX101: Overview of Creating a Course

The edX101 course is designed to provide a high-level overview of the course creation and delivery process using Studio and the edX LMS. It also highlights the extensive capabilities of the edX platform.

StudioX: Creating a Course with edX Studio

After you complete edX101, StudioX provides more detail about using Studio to create a course, add different types of content, and configure your course to provide an optimal online learning experience.

BlendedX: Blended Learning with edX

In BlendedX you explore ways to blend educational technology with traditional classroom learning to improve educational outcomes.

VideoX: Creating Video for the edX Platform

VideoX presents strategies for creating videos for course content and course marketing. The course provides step-by-step instructions for every stage of video creation, and includes links to exemplary sample videos created by edX partner institutions.

Documentation

Documentation for course teams is available from the docs.edx.org web page.

- Building and Running an edX Course is a comprehensive guide with concepts and procedures to help you build a course in Studio and then use the Learning Management System (LMS) to run a course.

  You can access this guide by selecting **Help** in Studio or from the instructor dashboard in the LMS.

- Using edX Insights describes the metrics, visualizations, and downloadable .csv files that course teams can use to gain information about student background and activity.

These guides open in your web browser. The left side of each page includes a **Search docs** field and links to the contents of that guide. To open or save a PDF version, select **v: latest** at the lower right of the page, then select **PDF**.

Note: If you use the Safari browser, be aware that it does not support the search feature for the HTML versions of the edX guides. This is a known limitation.

Email

To receive and share information by email, course team members can:

- Subscribe to announcements and other new topics in the edX Partner Portal or the Open edX Portal. For information about how to subscribe, see *Release Announcements through the Open edX Portal*.
- Join the openedx-studio Google group to ask questions and participate in discussions with peers at other edX partner organizations and edX staffers.

Wikis and Web Sites

The edX product team maintains public product roadmaps on *the Open edX Portal* and *the edX Partner Portal*.

The edX Partner Support site for edX partners hosts discussions that are monitored by edX staff.

1.2.6 Resources for Researchers

At each partner institution, the data czar is the primary point of contact for information about edX data. To set up a data czar for your institution, contact your edX partner manager.

Data for the courses on edx.org and edX Edge is available to the data czars at our partner institutions, and then used by database experts, statisticians, educational investigators, and others for educational research.

Resources are also available for members of the Open edX community who are collecting data about courses running on their sites and conducting research projects.

Documentation

The edX Research Guide is available on the docs.edx.org web page. Although it is written primarily for data czars and researchers at partner institutions, this guide can also be a useful reference for members of the Open edX community.

The *edX Research Guide* opens in your web browser, with a **Search docs** field and links to sections and topics on the left side of each page. To open or save a PDF version, select **v: latest** at the lower right of the page, and then select **PDF**.

Note: If you use the Safari browser, be aware that it does not support the search feature for the HTML versions of the edX guides. This is a known limitation.

Discussion Forums and Email

Researchers, edX data czars, and members of the global edX data and analytics community can post and discuss questions in our public research forum: the openedx-analytics Google group.

The edX partner portal also offers community forums, including a Research and Analytics topic, for discussions among edX partners.

Important: Please do not post sensitive data to public forums.

Data czars who have questions that involve sensitive data, or that are institution specific, can send them by email to data.support@edx.org with a copy to your edX partner manager.

Wikis

The edX Analytics team maintains the Open edX Analytics wiki, which includes links to periodic release notes and other resources for researchers.

The edx-tools wiki lists publicly shared tools for working with the edX platform, including scripts for data analysis and reporting.

1.2.7 Resources for Developers

Software engineers, system administrators, and translators work on extending and localizing the code for the edX platform.

Documentation

Documentation for developers is available from the edX Developer Documentation landing page.

GitHub

These are the main edX repositories on GitHub.

- The edx/edx-platform repo contains the code for the edX platform.
- The edx/edx-analytics-dashboard repo contains the code for edX Insights.
- The edx/configuration repo contains scripts to set up and operate the edX platform.

Additional repositories are used for other projects. Our contributor agreement, contributor guidelines and coding conventions, and other resources are available in these repositories.

Getting Help

The Getting Help page in the Open edX Portal lists different ways that you can ask, and get answers to, questions.

Wikis and Web Sites

The Open edX Portal is the entry point for new contributors.

The edX Engineering team maintains an open Confluence wiki, which provides insights into the plans, projects, and questions that the edX Open Source team is working on with the community.

The edx-tools wiki lists publicly shared tools for working with the edX platform, including scripts and helper utilities.

1.2.8 Resources for Open edX

Hosting providers, platform extenders, core contributors, and course staff all use Open edX. EdX provides release-specific documentation, as well as the latest version of all guides, for Open edX users. See the Open edX documentation page for a list of the documentation that is available.

1.3 edX Browser Support

Most current browsers will work on edX.org. For best performance, we recommend the latest versions of:

- Chrome
- Firefox

We also support the latest versions of:

- Microsoft Edge
- Safari

Note: If you use the Safari browser, be aware that it does not support the search feature for the guides on docs.edx.org. This is a known limitation.

CHAPTER

TWO

OPEN EDX ARCHITECTURE

The Open edX project is a web-based platform for creating, delivering, and analyzing online courses. It is the software that powers edx.org and many other online education sites.

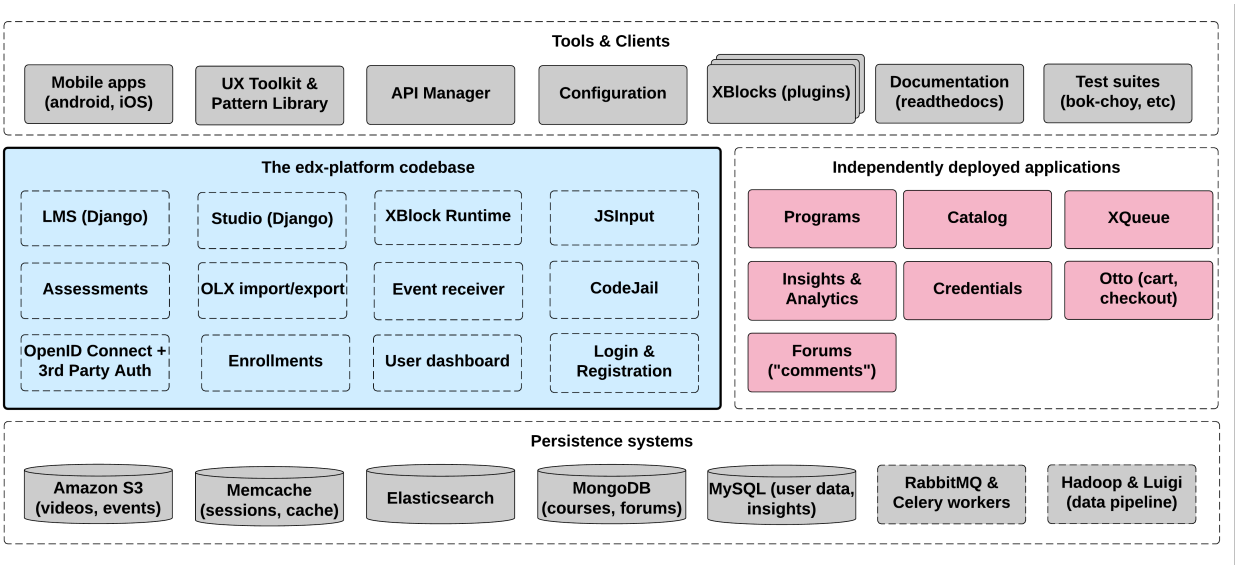
This page explains the architecture of the platform at a high level, without getting into too many details.

2.1 Overview

There are a handful of major components in the Open edX project. Where possible, these communicate using stable, documented APIs.

The centerpiece of the Open edX architecture is edx-platform, which contains the learning management and course authoring applications (LMS and Studio, respectively).

This service is supported by a collection of other autonomous web services called independently deployed applications (IDAs). Over time, edX plans to break out more of the existing edx-platform functions into new IDAs. This strategy will help manage the complexity of the edx-platform code base to make it as easy as possible for developers to approach and contribute to the project.



Almost all of the server-side code in the Open edX project is in Python, with Django as the web application framework.

2.2 Key Components

2.2.1 Learning Management System (LMS)

The LMS is the most visible part of the Open edX project. Learners take courses using the LMS. The LMS also provides an instructor dashboard that users who have the Admin or Staff role can access by selecting **Instructor**.

The LMS uses a number of data stores. Courses are stored in MongoDB, with videos served from YouTube or Amazon S3. Per-learner data is stored in MySQL.

As learners move through courses and interact with them, events are published to the analytics pipeline for collection, analysis, and reporting.

Front End

The Django server-side code in the LMS and elsewhere uses Mako for front-end template generation. The browser-side code is written primarily in JavaScript with some CoffeeScript as well (edX is working to replace that code with JavaScript). Parts of the client-side code use the Backbone.js framework, and edX is moving more of the code base to use that framework. The Open edX project uses Sass and the Bourbon framework for CSS code.

Course Browsing

The Open edX project provides a simple front page for browsing courses. The edx.org site has a separate home page and course discovery site that is not open source.

Course Structure

Open edX courses are composed of units called XBlocks. Anyone can write new XBlocks, allowing educators and technologists to extend the set of components for their courses. The edX platform also still contains several XModules, the precursors to XBlocks. EdX is working to rewrite the existing XModules as XBlocks and remove XModules from our code base.

In addition to XBlocks, there are a few ways to extend course behavior:

- The LMS is an LTI tool consumer. Course authors can embed LTI tools to integrate other learning tools into an Open edX course.
- Problems can use embedded Python code to either present the problem or assess the learner's response. Instructor-written Python code is executed in a secure environment called CodeJail.
- JavaScript components can be integrated using *JS Input*.
- Courses can be exported and imported using OLX (open learning XML), an XML- based format for courses.

2.2.2 Studio

Studio is the course authoring environment. Course teams use it to create and update courses. Studio writes its courses to the same Mongo database that the LMS uses.

2.2.3 Discussions

Course discussions are managed by an IDA called comments (also called forums). comments is one of the few non-Python components, written in Ruby using the Sinatra framework. The LMS uses an API provided by the comments service to integrate discussions into the learners' course experience.

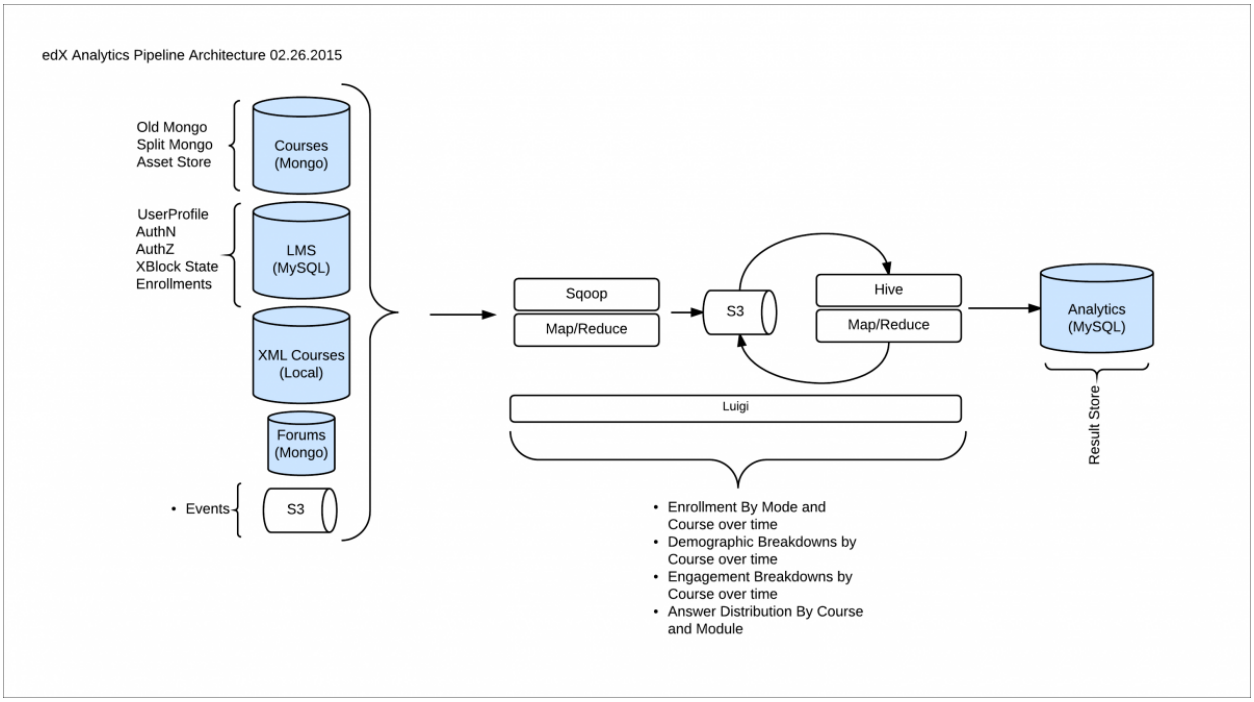
The comments service includes a notifier process that sends learners notifications about updates in topics of interest.

2.2.4 Mobile Apps

The Open edX project includes a mobile application, available for iOS and Android, that allows learners to watch course videos and more. EdX is actively enhancing the mobile app.

2.2.5 Analytics

Events describing learner behavior are captured by the Open edX analytics pipeline. The events are stored as JSON in S3, processed using Hadoop, and then digested, aggregated results are published to MySQL. Results are made available via a REST API to Insights, an IDA that instructors and administrators use to explore data that lets them know what their learners are doing and how their courses are being used.



2.2.6 Background Work

A number of tasks are large enough that they are performed by separate background workers, rather than in the web applications themselves. This work is queued and distributed using Celery and RabbitMQ. Examples of queued work include:

- Grading entire courses
- Sending bulk emails (with Amazon SES)
- Generating answer distribution reports
- Producing end-of-course certificates

The Open edX project includes an IDA called XQueue that can run custom graders. These are separate processes that run compute-intensive assessments of learners' work.

2.2.7 Search

The Open edX project uses Elasticsearch for searching in multiple contexts, including course search and the comments service.

2.2.8 Other Components

In addition to the components detailed above, the Open edX project also has services for other capabilities, such as one that manages e-commerce functions like order work flows and coupons.

CHAPTER

THREE

CONTRIBUTING TO OPEN EDX

3.1 Process for Contributing Code

Open edX is a massive project, and we would love you to help us build the best online education system in the world – we can't do it alone! However, the core committers on the project are also developing features and creating pull requests, so we need to balance reviewing time with development time. To help manage our time and keep everyone as happy as possible, we've developed this document that explains what core committers and other contributors can expect from each other. The goals are:

- Keep pull requests unblocked and flowing as much as possible, while respecting developer time and product owner prioritization.
- Maintain a high standard for code quality, while avoiding hurt feelings as much as possible.

3.1.1 Overview

Get in touch with us right now! We want to talk to you about your work as early as possible in your development cycle, ideally when you start designing your feature. This will help you so much, we promise: we'll help guide your work so you don't have to rearchitect it late in your development cycle. We'll help you understand the analytics, performance, test, accessibility, and other various requirements up front. We'll also be able to let you know if work you're planning duplicates work edX or others are doing. Finally, we'll let you know if your proposal, or parts of your proposal, would benefit from the Open edX Proposal (OEP) process.

You can get in touch with us using our community discussion forums.

You should get in touch with us regarding any work you intend to contribute upstream, including but not limited to these types of contributions.

- Core platform changes (changes to the edx-platform repo).
- Changes to any associated repos (edx-analytics, configuration, XBlock, etc.).
- A new XBlock you're writing that you intend to have run on edx.org.

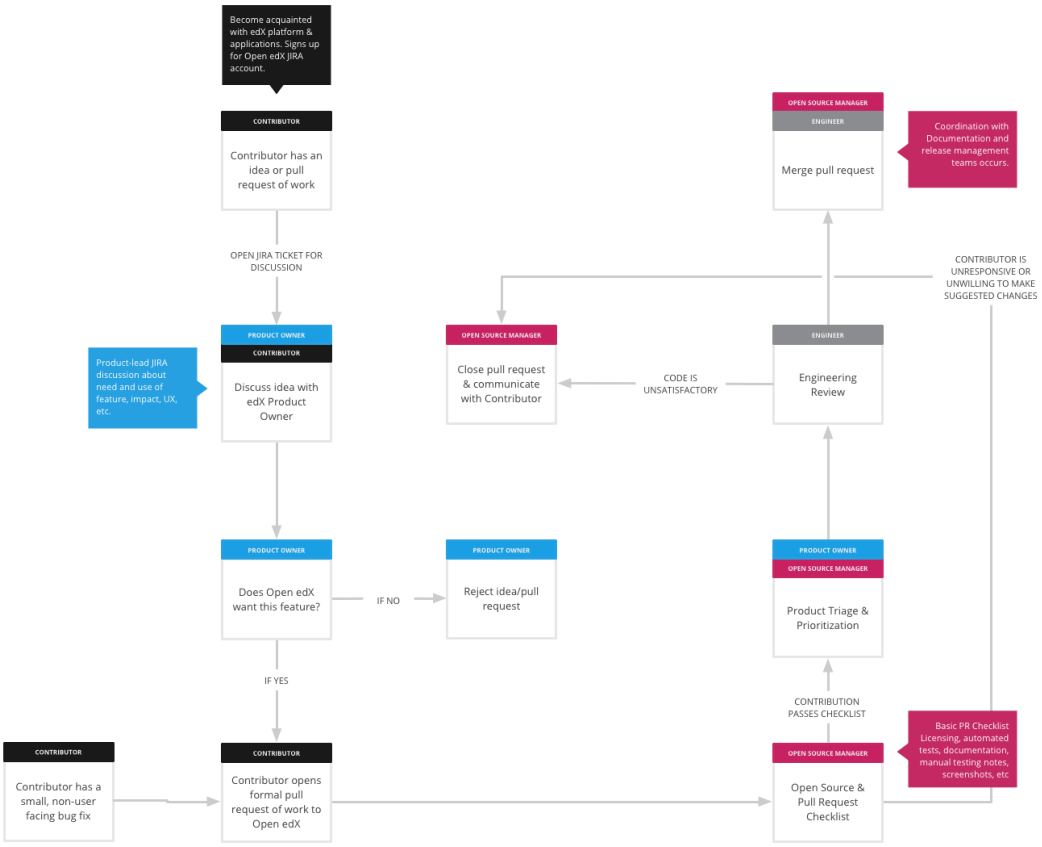
We do not need to review your code if you are writing a tool or customization for your own installation.

- An XBlock to be run on your own server, not intended to run on edx.org.
- LTI tools that are running on an external server.
- Code you are writing for your own installation, that won't be run on edx.org.

Make sure you've signed a contribution agreement before you submit code upstream to any edX owned repo. Please be sure you've at least skimmed the *contributor guidelines*, especially the *pull request cover letter* guidelines.

If you have any process questions along the way, please reach out to us on the community discussion forums.

Once you open your pull request, we'll need to review it. What types of review are necessary will be determined by the complexity of your pull request and whether or not you've spoken to us before you open your pull request. An overview of the process is here:



3.1.2 General Guidelines

Please follow these guidelines when writing code. Please note that not all of these may be required for your feature; reach out to us if you have any questions or concerns.

- Internationalization Coding Guidelines
- RTL UI Best Practices
- Accessibility Guidelines
- Analytics Guidelines
- Eventing Design and Review Process

For XBlocks you intend to install on edx.org, please also read the XBlock Review Guidelines.

3.1.3 Roles

People play different roles in the pull-request review process. Each role has different jobs and responsibilities.

Core Committer Can commit changes to an Open edX repository. Core committers are responsible for the quality of the code, and for supporting the code in the future. Core committers are also developers in their own right.

Product Owner Prioritizes the work of core committers.

Community Manager Helps keep the community healthy and working smoothly.

Contributor Submits pull requests for eventual committing to an Open edX repository.

If you are a *contributor* submitting a pull request, expect that it will take a few weeks before it can be merged. The earlier you can start talking with the rest of the Open edX community about the changes you want to make, before you even start changing code, the better the whole process will go.

Follow the guidelines in this document for a high-quality pull request: include a detailed description of your pull request when you open it on GitHub (we recommend using a *pull request cover letter* to guide your description), keep the code clear and readable, make sure the tests pass, be responsive to code review comments. Small pull requests are easier to review than large pull requests, so split up your changes into several small pull requests when possible – it will make everything go faster. See the full *contributor guidelines* for details of what to do and what to expect.

If you are a *product owner*, treat pull requests from contributors like feature requests from a customer. Keep the lines of communication open – if there are delays or unexpected problems, add a comment to the pull request informing the author of the pull request of what's going on. No one likes to feel like they're being ignored! More details are in the *product owner guidelines*.

If you are a *core committer*, allocate some time in your normal work schedule to review pull requests from other contributors. The community managers will make sure that these pull requests meet a basic standard for quality before asking you to spend time reviewing them. More details are in the *core committer guidelines*.

Feel free to read the other documentation specific to each individual role in the process, but you don't need to read everything to get started! If you're not sure where to start, check out the *contributor* documentation. Thanks for helping us grow the project smoothly! :)

3.2 Contributor

Before you make a pull request, it's a good idea to reach out to the Open edX community to discuss your ideas. There might well be someone else already working on the same change you want to make, and it's much better to collaborate than to submit incompatible pull requests. The Community Discussions page on the openedx.org website lists different ways you can ask, and answer, questions. The earlier you start the conversation, the easier it will be to make sure that everyone's on the right track.

It's also sometimes useful to submit a pull request even before the code is working properly, to make it easier to collect early feedback. To indicate to others that your pull request is not yet in a functional state, just prefix the pull request title with "(WIP)" (Work In Progress), or start the pull request as a draft on GitHub. Please include a link to a WIP pull request in any discussion threads you start.

Once you're ready to submit your changes in a pull request, check the following list of requirements to be sure that your pull request is ready to be reviewed:

1. Prepare a *pull request cover letter*. When you open your pull request, put your cover letter into the "Description" field on GitHub.
2. The code should be clear and understandable. Comments in code, detailed docstrings, and good variable naming conventions are expected. See the *Language Style Guidelines* for more details.

3. Commit messages should conform to OEP-51: Conventional Commits. This style categorizes commits to make them easier to understand.
4. The pull request should be as small as possible. Each pull request should encompass only one idea: one bug fix, one feature, etc. Multiple features (or multiple bug fixes) should not be bundled into one pull request. A handful of small pull requests is much better than one large pull request.
5. Structure your pull request into logical commits. "Fixup" commits should be squashed together. The best pull requests contain only a single, logical change – which means only a single, logical commit. You can squash your own commits, or the person merging the pull request may choose to squash them. They will do so by using the *Squash and merge* button of the GitHub interface to preserve the logical commits in the pull request, for forensic purposes when trying to diagnose a regression or understand a bug.
6. All code in the pull request must be compatible with Open edX's AGPL license. This means that the author of the pull request must sign a contributor's agreement, and all libraries included or referenced in the pull request must have compatible licenses.
7. All of the tests must pass. If a pull request contains a new feature, it should also contain new tests for that feature. If the pull request fixes a bug, it should also contain tests for that bug to be sure that it stays fixed. GitHub actions will verify this for your pull request, and point out any failing tests.
8. The author of the pull request should provide a test plan for manually verifying the change in this pull request. The test plan should include details of what should be checked, how to check it, and what the correct behavior should be. When it makes sense to do so, a good test plan includes a tarball of a small test course that has a unit which triggers the bug or illustrates the new feature.
9. For pull requests that make changes to the user interface, please include screenshots of what you changed. GitHub will allow you to upload images directly from your computer. In the future, the core committers will produce a style guide that contains more requirements around how pages should appear and how front-end code should be structured.
10. The pull request should contain some documentation for the feature or bug fix, either in a README file or in a comment on the pull request. A well- written description for the pull request may be sufficient.
11. The pull request should integrate with existing infrastructure as much as possible, rather than reinventing the wheel. In a project as large as Open edX, there are many foundational components that might be hard to find, but it is important not to duplicate functionality, even if small, that already exists.
12. The author of the pull request should be receptive to feedback and constructive criticism. The pull request will not be accepted until all feedback from reviewers is addressed. Once a core committer has reviewed a pull request from a contributor, no further review is required from the core committer until the contributor has addressed all of the core committer's feedback: either making changes to the pull request, or adding another comment explaining why the contributor has chosen not make any change based on that feedback.

It's also important to realize that you and the core committers may have different ideas of what is important in the codebase. The power and freedom of open source software comes from the fact that you can fork our software and make any modifications that you like, without permission from us. However, the core contributors are similarly empowered and free to decide what modifications to pull in from other contributors, and what not to pull in. While your code might work great for you on a small installation, it might not work as well on a large installation, have problems with performance or security, not be compatible with internationalization or accessibility guidelines, and so on. There are many, many reasons why the core committers may decide not to accept your pull request, even for reasons that are unrelated to the quality of your code change. However, if we do reject your pull request, we will explain why we aren't taking it, and try to suggest other ways that you can accomplish the same result in a way that we will accept.

3.2.1 Once A PR is Open

Once a pull request is open, automation will create a JIRA ticket in 2U's system to track review of your pull request. The JIRA ticket is a way for non-engineers (particularly, product owners) to understand your change and prioritize your pull request for team review.

If you open up your pull request with a solid description, following the *pull request cover letter* guidelines, the product owners will be able to quickly understand your change and prioritize it for review. However, they may have some questions about your intention, need, and/or approach that they will ask about. A community manager will ping you on GitHub to clarify these questions if they arise.

Once the product team has sent your pull request to the engineering teams for review, all technical discussion regarding your change will occur on GitHub, inline with your code.

3.2.2 Further Information

For further information on the pull request requirements, please see the following links:

- *Code Considerations*
- *Jenkins*
- *Code Coverage*
- *Code Quality*
- *EdX Python Style Guide*
- *EdX JavaScript Style Guide*
- *EdX Sass Style Guide*

3.3 Pull Request Cover Letter

For features with user-facing impact, please make sure you've also read the guidelines for *Contributing to the Documentation for Your Open Source Feature*.

When opening up a pull request, a cover letter template will be prepopulated for you. It has some items marked with **TODO**. These are indications of actions that you should take before (or as part of) submitting the pull request, and then can be deleted.

A good cover letter concisely answers as many of the following questions as possible. Not all pull requests will have answers to every one of these questions, which is okay!

- What JIRA ticket does this address (if any)? Please provide a link to the JIRA ticket representing the bug you are fixing or the feature discussion you've already had with the edX product owners.
- Who have you talked to at edX about this work? Design, architecture, previous PRs, course project manager, community discussions, etc. Please include links to relevant discussions.
- Why do you need this change? It's important for us to understand what problem your change is trying to solve, so please describe fully why you feel this change is needed.
- What components are affected? (LMS, Studio, a specific app in the system, etc.).
- What users are affected? For example, is this a new component intended for use in just one course, or is this a system wide change affecting all edX students?

- Test instructions for manual testing. When it makes sense to do so, a good test plan includes a tarball of a small test course that has a unit which triggers the bug or illustrates the new feature. Another option would be to provide explicit, numbered steps (ideally with screenshots!) to walk the reviewer through your feature or fix.
- Please provide screenshots for all user-facing changes.
- Indicate the urgency of your request. If this is a pull request for a course running or about to run on edx.org, we need to understand your time constraints. Good pieces of information to provide are the course(s) that need this feature and the date that the feature needed by.
- What are your concerns (the author's) about the PR? Is there a corner case you don't know how to address or some tests you aren't sure how to add? Please bring these concerns up in your cover letter so we can help!

3.3.1 Example Of A Good PR Cover Letter

Pull Request 4675 is one of the first edX pull requests to include a cover letter, and it is great! It clearly explains what the bug is, what system is affected (just the LMS), includes a tarball of a course that demonstrates the issue, and provides clear manual testing instructions.

Pull Request 4983 is another great example. This pull request's cover letter includes before and after screenshots, so the UX team can quickly understand what changes were made and make suggestions. Further, the pull request indicates how to manually test the feature and what date it is needed by.

3.4 Community Manager

Community managers handle the first part of the process of responding to pull requests, before they are reviewed by core committers. Community managers are responsible for monitoring the GitHub project so that they are aware of incoming pull requests. For each pull request, a community manager should:

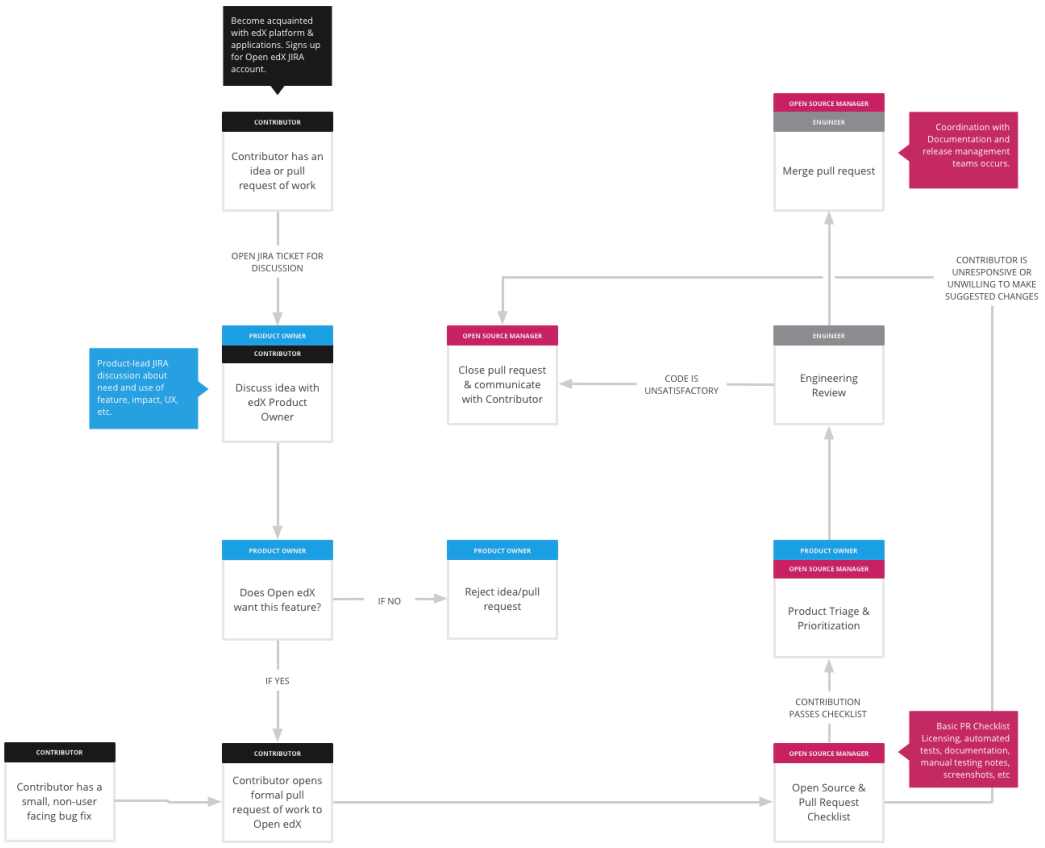
1. Read the description of the pull request to understand the idea behind it and what parts of the code it impacts. If the description is absent or unclear, inform the author that the pull request cannot be reviewed until the description is clearer. Guide them to the *pull request cover letter* guidelines.
2. Help the product team evaluate the idea behind the pull request. Is this something that Open edX wants? If you and the product owner(s) all believe that Open edX does not want this pull request, add a comment to the pull request explaining the reasoning behind that decision. Be polite, and remind them that they are welcome to fork the code and run their own fork on their own servers, without needing permission from edX. Try to suggest ways that they can build something that Open edX *does* want: for example, perhaps an API that would allow the contributor to build their own component separately. Then close the pull request.
3. Check that the author of the pull requests has submitted a contributor's agreement and completed any other necessary administrivia (our bot will make an automated comment if this is not properly in place). If not, inform author of problems and wait for them to fix it.
4. Once you've verified that the code change is not malicious, run a Jenkins job on the pull request and check the result. If there are failing tests (and they are real failures, not flaky tests), inform the author that the pull request cannot be reviewed until the tests are passing.
5. When all the tests pass, check the diff coverage and diff quality. If they are too low, inform the author of how to check these metrics, and ask the author to write unit tests to increase coverage and quality Diff quality should be 100%, and diff coverage should be at least 95% unless there are exceptional circumstances.
6. Skim the contents of the pull request and suggest obvious fixes/improvements to the pull request. Note that this is *not* a thorough code review – this is simply to catch obvious issues and low-hanging fruit. The point is to avoid interrupting core committers for trivial issues.

7. Ask the author of the pull request for a test plan: once this code is merged, how can we test that it's working properly? Whichever core committer merges this pull request will need to test it on a staging server before the code is deployed to production, so be sure that the test plan is clear enough for a core committer to follow.
8. If the PR includes any visual changes, or changes in user interaction, ask the author of the pull request to provide some screenshots. (For interaction changes, GIFs are awesome!) When a core committer starts reviewing the changes, it is often helpful to deploy the pull request to a sandbox server, so that the reviewer can click around and verify that the changes look good.
9. The core committers will put together a style guide. Pull requests that have visual/UX changes will be expected to respect this style guide – if they don't, point the author to the style guide and tell them to resubmit the pull request when it does.

At this point, the pull request is ready for code review. There are two different options: small PR review and large PR review. A PR is "small" if it can be read and understood in less than 15 minutes, including time spent context-switching, reading the description of the pull request, reading any necessary code context, etc. Typically, "small" PRs consist of fixing typos, improving documentation, adding comments, changing strings to unicode, marking strings that need to be translated, adding tests, and other chores. A "small" pull request doesn't modify the code that will be run in production in any meaningful way.

If the pull request is small, it can be reviewed immediately. If the community manager that is handling this pull request feels comfortable doing the code review, then he or she should do so rather than handing it off to a core committer. If not, he or she should move the JIRA ticket for the PR review into the "Awaiting Prioritization" state and add enough detail on the ticket for the product team to understand the size and scope of the changes. Inform the author that it might take a few days for the engineering team to review the PR.

If the pull request is not small, it will be handled by the full pull request process:



The community manager should:

- Make sure the pull request is ready for Product Review, if that has not yet happened. That means getting enough detail out of the contributor for the product owner to properly do a product review. Once this is done, move the JIRA ticket to the "Product Review" state.
- If questions arise from product owners during review, work with the contributor to get those questions answered before the next round of review.
- Once a PR has passed product review, do a first-round review of the PR with the contributor. That is, make sure quality and test coverage is up to par, and that the code generally meets our style guidelines. Once this has happened, move the ticket to the "Awaiting Prioritization" state.
- At each of these junctures, try to update the author with an estimate of how long the next steps will take. The product team will meet biweekly to review new proposals and prioritize PRs for team review. Direct the contributor to the JIRA ticket as well; the state of the JIRA ticket reflect the above diagram and can give a good sense of where in the process the pull request is.
- Once a PR has been prioritized for team review, ask the product owner for an estimate of how many sprints it will take for the pull request to be reviewed: if its more than one, try to push back and advocate for the contributor. However, the estimate is ultimately up to the product owner, and if he/she says it will really be more than one sprint, respect that.

- Add a comment to the pull request and inform the author that the pull request is queued to be reviewed. Give them an estimate of when the pull request will be reviewed: if you're not sure what to say, tell them it will be in two weeks. If the product owner has estimated that it will take more than one sprint before the pull request can be reviewed, direct the contributor to JIRA to monitor progress.

For determining which teams that the pull request impacts, use common sense – but in addition, there are a few guidelines:

- If any SASS files are modified, or any HTML in templates, include the UX (user experience) team.
- If any settings files or requirements files are modified, include the devops team.
- If any XModules are modified, include the blades team.
- If any logging events are modified, include the analytics team.
- Include the doc team on every contributor pull request that has a user-facing change.

Once the code review process has started, the community managers are also responsible for keeping the pull request unblocked during the review process. If a pull request has been waiting on a core committer for a few days, a community manager should remind the core committer to re-review the pull request. If a pull request has been waiting on a contributor for a few days, a community manager should add a comment to the pull request, informing the contributor that if they want the pull request merged, they need to address the review comments. If the contributor still has not responded after a few more days, a community manager should close the pull request. Note that if a contributor adds a comment saying something along the lines of "I can't do this right now, but I'll come back to it in X amount of time", that's fine, and the PR can remain open – but a community manager should come back after X amount of time, and if the PR still hasn't been addressed, he or she should warn the contributor again.

3.5 Product Owner

The product owner has two main responsibilities: approving user-facing features and improvements from a product point of view, and prioritizing pull request reviews.

When a contributor is interested in developing a new feature, or enhancing an existing one, they can engage in a dialogue with the product team about the feature: why it is needed, what does it do, etc. Product owners are expected to fully engage in this process and treat contributors like customers. If the idea is good but the implementation idea is poor, direct them to a better solution. If the feature is not something we can support at this time, provide a detailed explanation of why that is.

Approving work involves more than just giving the idea a go-ahead. There are a number of factors to consider.

- What level of support will edX provide? Unsupported, provisional, or supported?
- Will edx.org use the feature? Does it need configuration support?
- How much documentation is needed for the feature? Will edX write the documentation, or should the contributor provide it?
- Does the work require other review, such as user experience, design, accessibility, internationalization, training, or customer support?

The earlier in the process these other roles are involved, the better the process will work, and the better the final product will be.

A product owner is responsible for prioritizing pull requests from contributors, and keeping them informed when prioritization slips. Pull requests that are ready to be prioritized in the next sprint will have a "Awaiting Prioritization" label on their JIRA review tickets. At every product review meeting (which should happen each sprint), pull requests awaiting prioritization should either be included in the sprint for the appropriate team as a commitment to get the pull request reviewed, or the product owner must inform the author of the pull request that the pull request is still queued and

is not being ignored. Contributors should be treated as customers, and if their pull requests are delayed then they should be informed of that, just as a product owner would inform any customer when that customer's requests are delayed.

3.6 Core Committer

Core committers (or simply, committers) are responsible for reviewing pull requests from contributors. This happens once the pull request has passed through a community manager and been prioritized by a product owner. As much as possible, the code review process for community contributors should be identical to the process of reviewing a pull request from another committer: we're all part of the same community. However, there are a few ways that the process is different:

- The contributor cannot see when conflicts occur in the branch. These conflicts prevent the pull request from being merged, so you should ask the contributor to rebase their pull request, and point them to the documentation for doing so.
- Jenkins may not run on the contributor's pull request automatically. Be sure to start new Jenkins jobs for the PR as necessary – do not approve a pull request unless Jenkins has run, and passed, on the last commit in the pull request. If this contributor has already contributed a few good pull requests, that contributor can be added to the Jenkins whitelist, so that jobs are run automatically.
- The contributor may not respond to comments in a timely manner. This is not your concern: you can move on to other things while waiting. If there is no response after a few days, a community manager will warn the contributor that if the comments are not addressed, the pull request will be closed. (You can also warn the contributor yourself, if you wish.) Do not close the pull request merely because the contributor hasn't responded. If you think the pull request should be closed, inform the community managers, and they will handle it.

Each Scrum team should decide for themselves how to estimate stories related to reviewing external pull requests, and how to claim points for those stories, keeping in mind that an unresponsive contributor may block the story in ways that the team can't control. When deciding how many contributor pull request reviews to commit to in the upcoming iteration, teams should plan to spend about two hours per week per developer on the team – larger teams can plan to spend more time than smaller teams. For example, a team with two developers should plan to spend about four hours per week on pull request review, while a team with four developers should plan to spend about eight hours per week on pull request review – these hours can be spread out among multiple developers, or one developer can do all the review for the whole team in that iteration. However, this is just a guideline: the teams can decide for themselves how many contributor pull request reviews they want to commit to.

Once a pull request from a contributor passes all required code reviews, a core committer will need to merge the pull request into the project. The core committer who merges the pull request will be responsible for verifying those changes on the staging server prior to release, using the manual test plan provided by the author of the pull request.

In addition to reviewing contributor requests as part of sprint work, core committers should expect to spend about one hour per week doing other tasks related to the open source community: reading/responding to questions in the Community Discussions , disseminating information about what edX is working on, and so on.

3.6.1 Review Comments Terminology

In order to expedite the review process and to have a clear and mutual understanding between reviewers and contributors, the following terminology should be used when submitting comments on a PR:

- **Must** - A comment of type "Must" indicates the reviewer feels strongly about their requested change to the code and feels the PR should not be merged unless their concern is satisfactorily addressed.
- **Opt(ional)** - A comment of type "Optional" indicates the reviewer strongly favors their suggestion, but may be agreeable to the current behavior, especially with a persuasive response.

- **Nit(pick)** - A comment of type "Nitpick" indicates the reviewer has a minor criticism that *might* not be critical to address, but considers important to share in the given context. Contributors should still seriously consider and weigh these nits and address them in the spirit of maintaining high quality code.
- **FYI** - A comment of type "FYI" is a related side comment that is informative, but with the intention of having no required immediate action.

As an example, the following PR comment is clearly categorized as Optional:

```
"Optional:  Consider reducing the high degree of connascense in this code by using
keyword arguments."
```

Note: It is possible that after further discussion and review, the reviewer chooses to amend their comment, thereby changing its severity to be higher or lower than what was originally set.

3.7 Code Considerations

This is a checklist of all of the things that we expect developers to consider as they are building new, or modifying existing, functionality.

3.7.1 Operational Impact

- Are there new points in the system that require operational monitoring?
  - External system that you now depend on (Mathworks, SoftwareSecure, CyberSource, etc…)
  - New reliance on disk space?
  - New stand process (workers? elastic search?) that need to always be available?
  - A new queue that needs to be monitored for dequeueing
  - Bulk Email –> Amazon SES, Inbound queues, etc…
- Am I building a feature that will have impact on the performance of the system? Keep in mind that Open edX needs to support hundreds of thousands if not millions of students, so be careful that you code will work well when the numbers get large.
  - Deep Search

  - Grade Downloads
- Are reasonable log messages being written out for debugging purposes?
- Will this new feature easily start up in the Vagrant image?
- Do we have documentation for how to start up this feature if it has any new startup requirements?
- Are there any special directories/file system permissions that need to be set?
- Will this have any impact to the CDN related technologies?
- Are we pushing any extra manual burden on the Operations team to have to provision anything new when new courses launch? when new schools start? etc....
- Has the feature been tested using a production configuration with vagrant?

See also: *Deploy a New Service*.

3.7.2 Documentation/Training/Support

- Is there appropriate documentation in the context of the product for this feature, in the form of labels, on-screen help, or mouse over hints? In Studio, each page has a context-sensitive link to the edX documentation. If users will need more information to understand how to set up and use the feature, how can we get that information to the users?

  For more information about the types of information that technical writers typically try to find out when they document a new feature, see *Contributing to the Documentation for Your Open Source Feature*.
- Is this feature big enough that we need to have a session with stakeholders to introduce this feature BEFORE we release it? (PMs, Support, etc...)
  - Paid Certificates
- Do I have to give some more information to the Escalation Team so that this can be supported?
- Did you add an entry to CHANGELOG?
- Did you write/edit docstrings for all of your modules, classes, and functions?

3.7.3 Development

- Did you consider a reasonable upgrade path?
- Is this a feature that we need to slowly roll out to different audiences?
  - Bulk Email
- Have you considered exposing an appropriate amount of configuration options in case something happens?
- Have you considered a simple way to "disable" this feature if something is broken?
  - Centralized Logging
- Will this feature require any security provisioning?
  - Which roles use this feature? Does it make sense to ensure that only those roles can see this feature?
  - Assets in the Studio Library
- Did you ensure that any new libraries are added to appropriate provisioning scripts and have been checked by OSCM for license appropriateness?
- Is there an open source alternative?

- Are we locked down to any proprietary technologies? (AWS, . . .)
- Did you consider making APIs so that others can change the implementation if applicable?
- Did you consider Internationalization (I18N) and Localization (L10N)?
- Did you consider Accessibility (A11y)?
- Will your code work properly in workers?
- Have you considered the large-scale modularity of the code? For example, xModule and XBlock should not use Django features directly.

3.7.4 Testing

- Did you make sure that you tried boundary conditions?
- Did you try Unicode input/data?
  - The name of the person in paid certificates
  - The name of the person in bulk email
  - The body of the text in bulk email
  - etc.
- Did you try funny characters in the input/data? (~!@#$%^&*()';/.,<>, etc. . .)
- Have you done performance testing on this feature? Do you know how much performance is good enough?
- Did you ensure that your functionality works across all supported browsers?
- Do you have the right hooks in your HTML to ensure that the views can be automated?
- Are you ready if this feature has 10 times the expected usage?
- What happens if an external service does not respond or responds with a significant delay?
- What are possible failure modes? Do your unit tests exercise these code paths?
- Does this change affect templates and/or JavaScript? If so, are there Selenium tests for the affected page(s)? Have you tested the affected page(s) in a sandbox?

3.7.5 Analytics

- Are learning analytics events being recorded in an appropriate way?
  - Do your events use a descriptive and uniquely enough event type and namespace?
  - Did you ensure that you capture enough information for the researchers to benefit from this event information?
  - Is it possible to reconstruct the state of your module from the history of its events?
  - Has this new event been documented so that folks downstream know how to interpret it?
  - Are you increasing the amount of logging in any major way?
- Are you sending appropriate/enough information to MixPanel, Google Analytics, Segment?

3.7.6 Collaboration

- Are there are other teams that would benefit from knowing about this feature?
  - Forums/LMS - email
- Does this feature require a special broadcast to external teams as well?

3.7.7 Open Source

- Can we get help from the community on this feature?
- Does the community know enough about this?

3.7.8 UX/Design/Front End Development

- Did you make sure that the feature is going to pass *edX Accessibility Guidelines*?
- Did you make sure any system/instructional text is I18N ready?
- Did you ensure that basic functionality works across all supported browsers?
- Did you plan for the feature's UI to degrade gracefully (or be progressively enhanced) based on browser capability?
- Did you review the page/view under all browser/agent conditions - viewport sizes, images off, .css off?
- Did you write any HTML with ideal page/view semantics in mind?
- When writing HTML, did you adhere to standards/conventions around class/id names?
- When writing Sass, did you follow OOCSS/SMACSS philosophy ([1],[2],[3]), variable/extend organization and naming conventions, and UI abstraction conventions?
- When writing Sass, did you document any new variables, extend-based classes, or mixins?
- When writing/adding JavaScript, did you consider the asset pipeline and page load timeline?
- When writing JavaScript, did you note what code is for prototyping vs. production?
- When adding new templates, views, assets (Sass, images, plugins/libraries), did you follow existing naming and file architecture conventions?
- When adding new templates, views, assets (Sass, images, plugins/libraries), did you add any needed documentation?
- Did you use templates and good Sass architecture to keep DRY?
- Did we document any aspects about the feature (flow, purpose, intent) that we or other teams will need to know going forward?

[1] http://smacss.com/
[2] http://thesassway.com/intermediate/avoid-nested-selectors-for-more-modular-css
[3] http://ianstormtaylor.com/oocss-plus-sass-is-the-best-way-to-css/

3.7.9 Contributing to the Documentation for Your Open Source Feature

Thank you for making a contribution to Open edX. To help ensure the widest possible adoption for your contribution, it should have an appropriate level of documentation. For features with user-facing changes, additions to the edX documentation set might be needed to help different types of users understand and use it successfully.

You can use the questions that follow as guidelines for providing in-depth information about a change to the edX code base. The edX documentation team typically tries to answer questions like these for every new feature.

Your pull request ("PR") cover letter might already include some, or all, of this information, but we encourage you to consider each of these questions to be sure that you have provided thorough context and detail.

The edX documentation set is created using RST files and Sphinx. If you want to contribute documentation directly, you are welcome to make revisions and additions to the files in the edX documentation team's GitHub repository. If you have questions, please contact us at docs@edx.org.

1. What problem or lack of functionality do users experience that made you decide to make this contribution?
2. How does your feature or revision address that problem? Consider providing one or more use cases.
3. Who is affected by your contribution, and in what ways? Please provide one or more screen captures.
    - Will the course team have access to a new tool or page in Studio, or see changes or additions to the Studio user interface?
    - How will learners experience the change in the course content? What learning outcomes can be expected?
    - How will course team members experience the change in the LMS, on the Instructor Dashboard as well as in the course content?
    - What questions are researchers likely to ask about student interaction with the feature? Will researchers need information about new or changed tracking log events, SQL tables, or JSON files?
    - Does this feature include tools for developers, such as a new API or changed or updated API endpoints?
4. Does your contribution affect any existing problem types or the video player? The events emitted by these features are used by Open edX Insights and by researchers to measure learner performance and engagement.
    - Performance analytics: What effect does your change have on existing data, reports, and metrics for student performance? Have you added reports or metrics?
    - Engagement analytics: What effect does your change have on existing data, reports, and metrics for student engagement? Have you added reports or metrics?
5. Are there any prerequisites?
    - Does a system administrator need to set a feature flag, grant permissions, set up a user account, configure integration with a third party tool, or perform any other installation or configuration steps? If so, be sure to provide those steps.
    - Do any Advanced Setting policy keys need to be added or changed in Studio? If so, be sure to provide an example of the syntax needed.
    - Is a particular course role needed to set up or use the feature? Some examples are discussion moderator, beta tester, and admin.
    - Is specialized background knowledge necessary? Examples are familiarity with, or authorization to access, other on campus systems or third party tools.
6. How will each affected audience (particularly system administrators, course teams, and learners) use the feature? Consider describing the workflow and referencing screen captures.

CHAPTER

FOUR

EXTENDING THE EDX PLATFORM

4.1 Options for Extending the edX Platform

There are several options for extending the Open edX Platform to provide useful and innovative educational content in your courses.

This section of the developers' documentation lists and explains the different ways to extend the platform, starting with the following table.

	Custom JavaScript Applications*	LTI	External Graders	XBlocks	Platform Customization
Development Cost	Low	Low	Medium	Medium	High
Language	JavaScript	Any	Any	Python	Python
Development Environment Needed	No	No	Yes	Yes	Yes
Self-hosting Needed	No	Yes	Yes	No	No
Need edX Involvement	No	No	Yes	Yes	Yes
Clean UI Integration	Yes	No (see LTI)	Yes	Yes	Yes
Mobile enabled	Possibly	Possibly	Yes	Yes	Yes
Server Side Grading	Possibly (See JavaScript)	Yes	Yes	Yes	Yes
Usage Data	No (See JavaScript)	No	Limited	Yes	Yes
Provision in Studio	No	No	No	Yes	No
Privacy Loss Compared to Hosting Open edX	No	Possibly	Possibly	No	No

(*) *Custom JavaScript Applications*

4.2 Integrating XBlocks with edx-platform

The edX LMS and Studio have several features that are extensions of the core XBlock libraries (https://xblock.readthedocs.io). These features are listed below.

- *LMS*
- *Studio*
- *Testing*
- *Deploying your XBlock*

You can also render an individual XBlock in HTML with the *XBlock URL*.

4.2.1 LMS

Runtime Features

These are properties and methods available on `self.runtime` when a view or handler is executed by the LMS.

- anonymous_student_id: An identifier unique to the student in the particular course that the block is being executed in. The same student in two different courses will have two different ids.
- publish(block, event_type, event): Emit events to the surrounding system. Events are dictionaries that can contain arbitrary data. XBlocks can publish events by calling `self.runtime.publish(self, event_type, event)`. The `event_type` parameter enables downstream processing of the event since it uniquely identifies the schema. This call will cause the runtime to save the event data in the application event stream. XBlocks should publish events whenever a significant state change occurs. Post-hoc analysis of the event stream can yield insight about how the XBlock is used in the context of the application. Ideally interesting state of the XBlock could be reconstructed at any point in history through careful analysis of the event stream.

In the future, these are likely to become more formal XBlock services (one related to users, and the other to event publishing).

Class Features

These are class attributes or functions that can be provided by an XBlock to customize behavior in the LMS.

- student_view (XBlock view): This is the view that will be rendered to display the XBlock in the LMS. It will also be used to render the block in "preview" mode in Studio, unless the XBlock also implements author_view.
- has_score (class property): True if this block should appear in the LMS progress page.
- get_progress (method): See documentation in x_module.py:XModuleMixin.get_progress.
- icon_class (class property): This can be one of (`other`, `video`, or `problem`), and determines which icon appears in edx sequence headers. There is currently no way to provide a different icon.

Grading

To participate in the course grade, an XBlock should set `has_score` to `True`, and should `publish` a `grade` event whenever the grade changes. The `grade` event is a dictionary of the following form.

```
{
    'value': <number>,
    'max_value': <number>,
    'user_id': <number>,
}
```

The grade event represents a grade of `value/max_value` for the current user. The `user_id` field is optional, the currently logged in user's ID will be used if it is omitted.

Restrictions

A block cannot modify the value of any field with a scope where the `user` property is `UserScope.NONE`.

4.2.2 Studio

Class Features

- studio_view (XBlock.view): The view used to render an editor in Studio. The editor rendering can be completely different from the LMS student_view, and it is only shown when the author selects "Edit".
- author_view (XBlock.view): An optional view of the XBlock similar to student_view, but with possible inline editing capabilities. This view differs from studio_view in that it should be as similar to student_view as possible. When previewing XBlocks within Studio, Studio will prefer author_view to student_view.
- non_editable_metadata_fields (property): A list of `xblock.fields.Field` objects that should not be displayed in the default editing view for Studio.

Restrictions

A block cannot modify the value of any field with a scope where the `user` property is not `UserScope.NONE`.

Testing

These instructions are temporary. Once XBlocks are fully supported by edx-platform (both the LMS and Studio), installation and testing will be much more straightforward.

To enable an XBlock for testing in your devstack (https://github.com/openedx/configuration/wiki/edX-Developer-Stack), follow these steps.

1. Install your block.

   ```
   $ vagrant ssh
   vagrant@precise64:~$ sudo -u edxapp /edx/bin/pip.edxapp install /path/to/your/block
   ```

2. Enable the block.

   1. In `edx-platform/lms/envs/common.py`, uncomment:

```
# from xmodule.x_module import prefer_xmodules
# XBLOCK_SELECT_FUNCTION = prefer_xmodules
```

2. In `edx-platform/cms/envs/common.py`, uncomment:

```
# from xmodule.x_module import prefer_xmodules
# XBLOCK_SELECT_FUNCTION = prefer_xmodules
```

3. Add the block to your courses' advanced settings in Studio.
    1. Log in to Studio, and open your course
    2. Settings -> Advanced Settings
    3. Change the value for the key `"advanced_modules"` to `["your-block"]`
4. Add your block into your course.
    1. Edit a unit
    2. Advanced -> your-block

Note the name `your-block` used in Studio must exactly match the key you used to add your block to your `setup.py` `entry_points` list. (If you are still discovering XBlocks and simply used the `workbench-make-new.py` script as described in the Open edX XBlock Tutorial, look in the `setup.py` file that was created.)

4.2.3 Deploying Your XBlock

To deploy your block to your own hosted version of edx-platform, you need to install it into the virtualenv that the platform is running out of, and add to the list of `ADVANCED_COMPONENT_TYPES` in `edx-platform/cms/djangoapps/contentstore/views/component.py`.

4.2.4 Rendering XBlocks with the XBlock URL

The XBlock URL supports HTML rendering of an individual XBlock without the user interface of the LMS.

To use the XBlock URL and return the HTML rendering of an individual XBlock, you use the following URL path for an XBlock on an edX site.

`https://{host}/xblock/{usage_id}`

Finding the `usage_id`

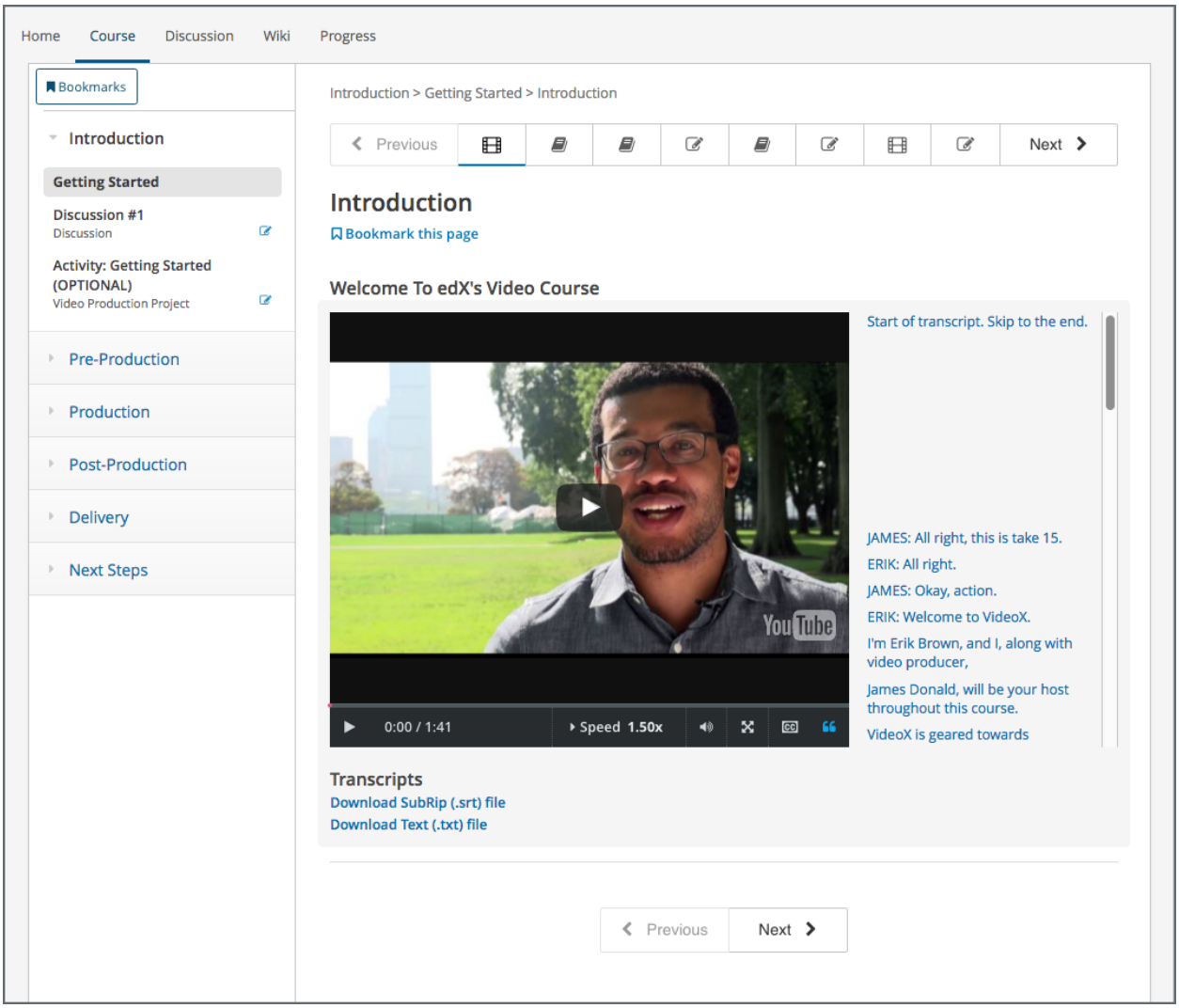
The `usage_id` is the unique identifier for the problem, video, text, or other course content component, or for sequential or vertical course container component. There are several ways to find the `usage_id` for an XBlock in the LMS, including viewing either the staff debug info or the page source. For more information, see Finding the Usage ID for Course Content.

Example XBlock URLs

For example, a video component in the "Creating Video for the edX Platform" course on the edx.org site has the following URL.

```
https://courses.edx.org/courses/course-v1:edX+VideoX+1T2016/courseware/
ccc7c32c65d342618ac76409254ac238/1a52e689bcec4a9eb9b7da0bf16f682d/
```

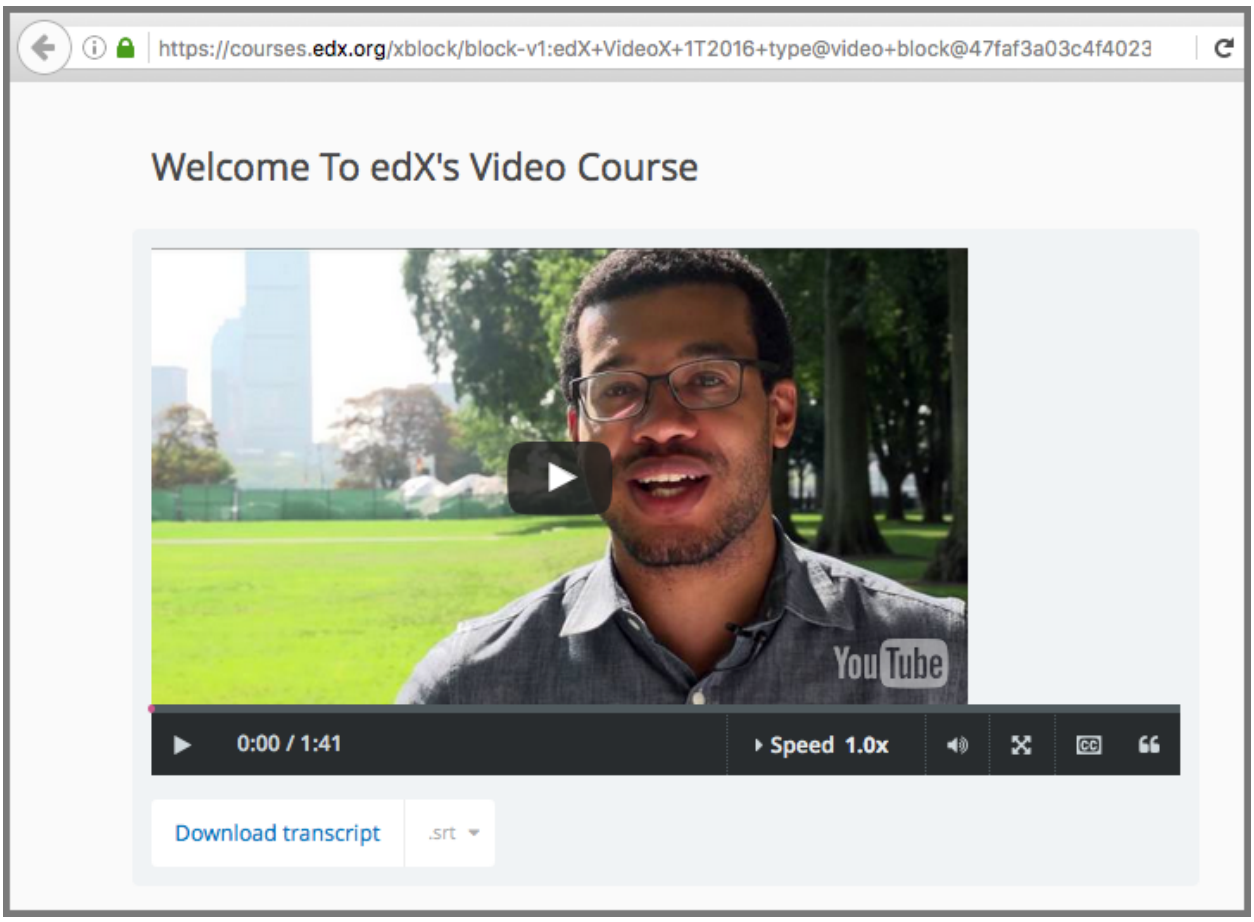
This video component appears as follows in the LMS.



To construct the XBlock URL for the same video component, you obtain its `usage_id` and then use the following URL format.

```
https://courses.edx.org/xblock/block-v1:edX+VideoX+1T2016+type@video+block@47faf3a03c4f4023b187528c2593
```

When you use this URL, the video component appears in your browser as follows.



For courses created prior to October 2014, the `usage_id` begins with `i4x://`, as in the following example.

`https://courses.edx.org/xblock/i4x://edX/DemoX.1/problem/47bf6dbce8374b789e3ebdefd74db332`

4.3 Custom JavaScript Applications

4.3.1 Overview

You can include custom JavaScript applications (also called custom JavaScript problems or JS input problems) in a course. You add the application directly into edX Studio.

When you create a JavaScript application, Studio embeds the problem in an inline frame (HTML `iframe` tag) so that learners can interact with it in the LMS.

See the following sections for more information:

- *Grading Options for Custom JavaScript Applications*
- *Use a JavaScript Application Without Grading*
- *Use a JavaScript Application for a Summative Assessment*
- *Grade the Student Response with Python*
- *XML for Custom JavaScript Applications*

See *The Custom JavaScript Display and Grading Example Template* for information about the template application built in to edX Studio.

Course teams should see the following sections of the Building and Running an edX Course guide.

- Custom JavaScript Display and Grading
- Establishing a Grading Policy

The rest of this section provides more information for developers who are creating JavaScript applications for courses on the edX platform.

Note: This section assumes proficiency with JavaScript and with how problems are constructed in edX Studio. If you intend to grade learners' interactions with your JavaScript application, you must also be proficient with Python.

4.3.2 Grading Options for Custom JavaScript Applications

When using a JavaScript application in your course content, you have three options.

1. A JavaScript application that visually demonstrates a concept or process. The application would not require learner interaction, and learners would not be graded.
2. A JavaScript application that requires learner interaction but does not grade performance. Referred to as a formative assessment, such an application provides feedback to learners based on their interactions.
3. A JavaScript application that requires and grades learner interaction. Referred to as a summative assessment, such an application can be used to evaluate learning against a standard. To use the JavaScript application as a summative assessment and have learner performance integrated into the edX grading system, you must also use basic Python code in the component.

These options are explained through examples below.

4.3.3 Use a JavaScript Application Without Grading

The simplest option is to use JavaScript to show content to learners, and optionally to provide feedback as a formative assessment.

1. In edX Studio, upload an HTML file that contains the JavaScript you want to show learners.
2. Copy the **Embed URL** of the file.
3. Create a Custom JavaScript Display and Grading problem. The template for the problem contains the definition for a sample JavaScript application that requires and grades learner interaction.
4. Edit the XML of the component to remove grading information and refer to the HTML file you uploaded:

```
<customresponse>
    <jsinput
     width="width needed to display your application"
     height="height needed to display your application"
     html_file="Embed URL of the HTML file"
     sop="false"/>
</customresponse>
```

For example:

```
<customresponse>
    <jsinput
     width="400"
     height="400"
     html_file="/static/electrol_demo.html"
     sop="false"/>
</customresponse>
```

4.3.4 Use a JavaScript Application for a Summative Assessment

To use a JavaScript Application for a summative assessment and have learner results calculated by the edX grading system, you must:

- Include these required functions in the JavaScript application.
  - *getState() Function*
  - *setState() Function*
  - *getGrade() Function*
- Reference functions in the problem XML.
- *Grade the Student Response with Python*.

getState() Function

Your application must contain a `getState()` function that returns the state of all objects as a JSON string.

The `getState()` function retrieves the state of objects in the application, so each learner experiences that application in its initial or last saved state.

The name of the `getState()` function must be the value of the `get_statefn` attribute of the `jsinput` element for the problem.

For example:

```
<customresponse cfn="vglcfn">
    <jsinput get_statefn="JSObject.getState"
        . . . .
```

setState() Function

Your application must contain a `setState()` function.

The `setState()` function is executed when the learner selects **Submit**.

The function saves application's state so that the learner can later return to the application and find it as he or she left it.

The name of the `setState()` function must be the value of the `set_statefn` attribute of the `jsinput` element for the problem.

For example:

```
<customresponse cfn="vglcfn">
    <jsinput set_statefn="JSObject.setState"
        . . . .
```

getGrade() Function

Your application must contain a `getGrade()` function.

The `getGrade()` function is executed when the learner selects **Submit**. The `getState()` function must return the state of objects on which grading is based as a JSON string.

The JSON string returned by `getGrade()` is used by the Python code in the problem to determine the learner's results, as explained below.

The name of the `getGrade()` function must be the value of the `gradefn` attribute of the `jsinput` element for the problem.

For example:

```
<customresponse cfn="vglcfn">
    <jsinput gradefn="JSObject.getGrade"
        . . . .
```

4.3.5 Grade the Student Response with Python

To grade a learner's interaction with your JavaScript application, you must write Python code in the problem. When a learner selects **Submit**, the Python code parses the JSON string returned by the application's `getGrade()` function and determines if the learner's submission is correct or not.

Note: Grading for JavaScript applications supports determining if a learner's submission is correct or not. You cannot give partial credit with JavaScript applications.

In the Python code, make sure you follow these guidelines.

- Enclose all code in a `script` element of type `loncapa/python`.
- Import `json`
- Define a function that is executed when the learner selects **Submit**, and that meets the following requirements.
  - Is placed before the `customresponse` element that defines the problem.
  - By default is named `vglcfn`
  - Has two parameters: `e` for the submission event, and `ans`, which is the JSON string returned by the JavaScript function `getGrade()`.
  - Must return `True` if the learner's submission is correct, or `False` if it is incorrect.

The structure of the Python code in the problem is shown in this example.

```
<problem>
    <script type="loncapa/python">
        import json
        def vglcfn(e, ans):
```

(continues on next page)

(continued from previous page)

```
            """
            Code that parses ans and returns True or False
            """
    </script>
    <customresponse cfn="vglcfn">
    . . . .
</problem>
```

4.3.6 XML for Custom JavaScript Applications

The problem component XML that you define in Studio to provide learners with a JavaScript application has the following structure.

```
<problem>
    <!-- Optional script tag for summative assessments -->
    <script type="loncapa/python">
        import json
        def vglcfn(e, ans):
            """
            Code that parses ans and returns True or False
            """
    </script>
    <customresponse cfn="vglcfn">
        <jsinput
            gradefn="JSObject.getGrade"
            get_statefn="JSObject.getState"
            set_statefn="JSObject.setState"
            width="100%"
            height="360"
            html_file="/static/file-name.html"
            sop="false"/>
    </customresponse>
</problem>
```

jsinput attributes

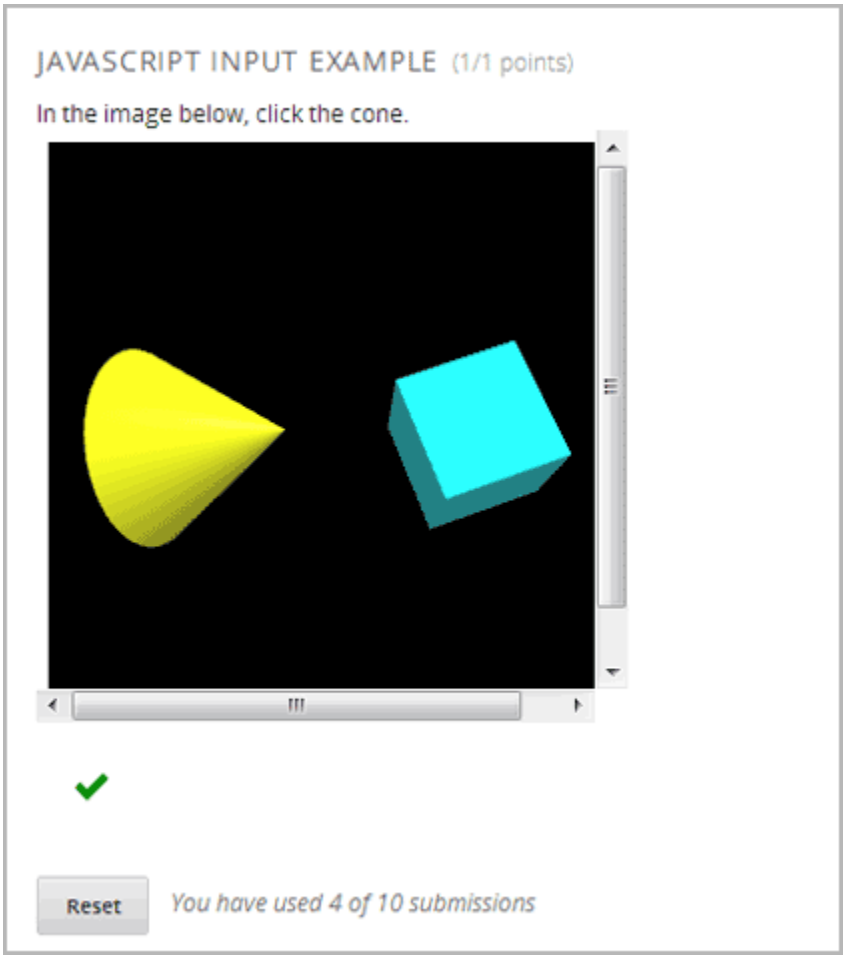
The following table describes the attributes of the `jsinput` element.

Attribute	Description	Example
gradefn	The function in your JavaScript application that returns the state of the objects to be evaluated as a JSON string.	`JSObject.getGrade`
get_statefun	The function in your JavaScript application that returns the state of the objects.	`JSObject.getState`
set_statefun	The function in your JavaScript application that saves the state of the objects.	`JSObject.setState`
initial_state	A JSON string representing the initial state, if any, of the objects.	‘{“selectedObjects”:{“cube”:true,”cylinder”:fal
width	The width of the iframe in which your JavaScript application will be displayed, in pixels.	400
height	The height of the iframe in which your JavaScript application will be displayed, in pixels.	400
html_file	The name of the HTML file containing your JavaScript application that will be loaded in the iframe.	/static/webGLDemo.html
sop	The same-origin policy (SOP), meaning that all elements have the same protocol, host, and port. To bypass the SOP, set to `true`.	false

4.4 The Custom JavaScript Display and Grading Example Template

As referred to in course team documentation, there is a built-in template in edX Studio that uses a sample JavaScript application.

This sample application has learners select two different shapes, a cone and a cube. The correct state is when the cone is selected and the cube is not selected.



You can download files for that application. You must upload these files in Studio to use them in a problem.

The following information steps through this example to demonstrate how to apply the guidelines in *Custom JavaScript Display and Grading*.

4.4.1 Example getState() Function

In this example, the `state` variable is initialized for the cylinder and cube in the `WebGLDemo.js` file.

```
var state = {
    'selectedObjects': {
        'cylinder': false,
        'cube': false
    }
}
```

User interactions toggle the `state` values of the cylinder and cube between `true` and `false`.

The `getState()` function in the sample application returns the state as a JSON string.

```
function getState() {
    return JSON.stringify(state);
 }
```

4.4.2 Example setState() Function

In this example, when a learner selects **Submit**, the `state` variable is saved so that the learner can later return to the application and find it in the same state.

```
function setState() {
    stateStr = arguments.length === 1 ? arguments[0] : arguments[1];
    state = JSON.parse(stateStr);
    updateMaterials();
}
```

The `updateMaterials()` function called by `setState()` updates the state of the cylinder and cone with the user's current selections:

```
function updateMaterials() {
    if (state.selectedObjects.cylinder) {
        cylinder.material = selectedMaterial;
    }
    else {
        cylinder.material = unselectedMaterial;
    }

    if (state.selectedObjects.cube) {
        cube.material = selectedMaterial;
    }
    else {
        cube.material = unselectedMaterial;
    }
}
```

4.4.3 Example getGrade() function

In this example, when a learner selects **Submit**, the `getGrade()` function returns the selected objects.

```
function getGrade() {
    return JSON.stringify(state['selectedObjects']);
}
```

The returned JSON string is then used by the Python code defined in the problem to determine if correct objects were selected or not, and to return a result.

4.4.4 Grade the Student Response

The following is the Python function `vglcfn` in the sample application:

```
<script type="loncapa/python">
import json
def vglcfn(e, ans):
    """
    par is a dictionary containing two keys, "answer" and "state"
    The value of answer is the JSON string returned by getGrade
    The value of state is the JSON string returned by getState
```

(continues on next page)

(continued from previous page)

```
    """
    par = json.loads(ans)
    # We can use either the value of the answer key to grade
    answer = json.loads(par["answer"])
    return answer["cylinder"] and not answer["cube"]
    """
    # Or we could use the value of the state key
    state = json.loads(par["state"])
    selectedObjects = state["selectedObjects"]
    return selectedObjects["cylinder"] and not selectedObjects["cube"]
    """
</script>
```

The `ans` parameter contains the JSON string returned by `getGrade()`. The value is converted to a Python Unicode structure in the variable `par`.

In the function's first option, object(s) the learner selected are stored in the `answer` variable. If the learner selected the cylinder and not the cube, the `answer` variable contains only `cylinder`, and the function returns `True`, which signifies a correct answer. Otherwise, it returns `False` and the answer is incorrect.

In the function's second option, the objects' states are retrieved. If the cylinder is selected and not the cube, the function returns `True`, which signifies a correct answer. Otherwise, it returns `False` and the answer is incorrect.

4.4.5 XML Problem Structure

The XML problem for the sample template is as follows.

```
<problem display_name="webGLDemo">
    <script type="loncapa/python">
        import json
        def vglcfn(e, ans):
            """
            par is a dictionary containing two keys, "answer" and "state"
            The value of answer is the JSON string returned by getGrade
            The value of state is the JSON string returned by getState
            """
            par = json.loads(ans)
            # We can use either the value of the answer key to grade
            answer = json.loads(par["answer"])
            return answer["cylinder"] and not answer["cube"]
            """
            # Or we could use the value of the state key
            state = json.loads(par["state"])
            selectedObjects = state["selectedObjects"]
            return selectedObjects["cylinder"] and not selectedObjects["cube"]
            """
    </script>
    <p>
        The shapes below can be selected (yellow) or unselected (cyan).
        Clicking on them repeatedly will cycle through these two states.
    </p>
    <p>
```

(continues on next page)

(continued from previous page)

```
        If the cone is selected (and not the cube), a correct answer will
        be generated after selecting "Submit". Selecting either "Submit"
        or "Save" will register the current state.
    </p>
    <customresponse cfn="vglcfn">
        <jsinput gradefn="WebGLDemo.getGrade"
            get_statefn="WebGLDemo.getState"
        set_statefn="WebGLDemo.setState"
        width="400"
        height="400"
        html_file="https://studio.edx.io/c4x/edX/DemoX/asset/webGLDemo.html"
        sop="false"/>
    </customresponse>
</problem>
```

CHAPTER

FIVE

TESTING

Testing is something that we take very seriously at edX: we even have a "test engineering" team at edX devoted purely to making our testing infrastructure even more awesome.

To find out more about our testing infrastructure, check out the testing.rst file on GitHub.

5.1 Jenkins

Jenkins is an open source continuous integration server. edX has a Jenkins installation specifically for testing pull requests to our open source software project, including edx-platform. Before a pull request can be merged, Jenkins must run all the tests for that pull request: this is known as a "build". If even one test in the build fails, then the entire build is considered a failure. Pull requests cannot be merged until they have a passing build.

5.1.1 Kicking Off Builds

Jenkins has the ability to automatically detect new pull requests and changed pull requests on GitHub, and it can automatically run builds in response to these events. We have Jenkins configured to automatically run builds for all pull requests from core committers; however, Jenkins will *not* automatically run builds for new contributors, so a community manager will need to manually kick off a build for a pull request from a new contributor.

The reason for this distinction is a matter of trust. Running a build means that Jenkins will execute all the code in the pull request. A pull request can contain any code whatsoever: if we allowed Jenkins to automatically build every pull request, then a malicious developer could make our Jenkins server do whatever he or she wanted. Before kicking off a build, community managers look at the code changes to verify that they are not malicious; this protects us from nasty people.

Once a contributor has submitted a few pull requests, they can request to be added to the Jenkins whitelist: this is a special list of people that Jenkins *will* kick off builds for automatically. If the community managers feel that the contributor is trustworthy, then they will grant the request, which will make future development faster and easier for both the contributor and edX. If a contibutor shows that they can not be trusted for some reason, they will be removed from this whitelist.

5.1.2 Failed Builds

Click on the build to be brought to the build page. You'll see a matrix of blue and red dots; the red dots indicate what section failing tests were present in. You can click on the test name to be brought to an error trace that explains why the tests fail. Please address the failing tests before requesting a new build on your branch. If the failures appear to not have anything to do with your code, it may be the case that the master branch is failing. You can ask your reviewers for advice in this scenario.

If the build says "Unstable" but passes all tests, you have introduced too many pep8 and pylint violations. Please refer to the documentation for *Code Quality* and clean up the code.

5.1.3 Successful Builds

If all the tests pass, the "Diff Coverage" and "Diff Quality" reports are generated. Click on the "View Reports" link on your pull request to be brought to the Jenkins report page. In a column on the left side of the page are a few links, including "Diff Coverage Report" and "Diff Quality Report". View each of these reports (making note that the Diff Quality report has two tabs - one for pep8, and one for Pylint).

Make sure your quality coverage is 100% and your test coverage is at least 95%. Adjust your code appropriately if these metrics are not high enough. Be sure to ask your reviewers for advice if you need it.

5.2 Code Coverage

We measure which lines of our codebase are covered by unit tests using coverage.py for Python and JSCover for Javascript.

Our codebase is far from perfect, but the goal is to steadily improve our coverage over time. To do this, we wrote a tool called diff-cover that will report which lines in your branch are not covered by tests, while ignoring other lines in the project that may not be covered. Using this tool, we can ensure that pull requests have a very high percentage of test coverage – and ideally, they increase the test coverage of existing code, as well.

To check the coverage of your pull request, just go to the top level of the edx-platform codebase and run:

```
$ paver coverage
```

This will print a coverage report for your branch. We aim for a coverage report score of 95% or higher. We also encourage you to write acceptance tests as your changes require.

5.3 Code Quality

We use a variety of tools to check for errors and vulerabilities, and to enforce a coding standard and coding style.

To check the quality of your pull request, go to the top level of the edx- platform codebase and run the following command.

```
$ paver run_quality
```

The following topics provide additional details on the tools that we use.

- *Clean Code*
- *Safe Code*

5.3.1 Clean Code

Here are the primary tools we use to keep our code clean.

- We use the pep8 tool to follow PEP-8 guidelines.
- We use pylint for static analysis and to uncover trouble spots in our code.

Our codebase is far from perfect, but the goal is to steadily improve its quality over time. To do this, we wrote a pypi package called diff-cover, which includes the tool diff-quality. The diff-quality tool reports on quality violations only on lines that have changed in a pull request. Using this tool, we can ensure that pull requests do not introduce new quality violations, and also clean up existing violations in the process of introducing other changes.

To run diff-quality along with our other quality based tools, go to the top level of the edx-platform codebase and run the following command.

```
$ paver run_quality
```

You can also use the `paver run_pep8` and `paver run_pylint` commands to run only pep8 or pylint.

This will print a report of the quality violations that your branch has made.

Although we try to be vigilant in resolving all quality violations, some Pylint violations are too challenging to resolve, so we opt to ignore them via use of a pragma. A pragma tells Pylint to ignore the violation in the given line. An example is:

```
self.assertEquals(msg, form._errors['course_id'][0])  # pylint: disable=protected-access
```

The pragma starts with a `#` two spaces after the end of the line. We prefer that you use the full name of the error (`pylint: disable=unused-argument` as opposed to `pylint: disable=W0613`), to make more clear what you are disabling in the line.

5.3.2 Safe Code

To keep our code safe from Cross Site Scripting (XSS) vulnerabilities, the XSS Linter is also run as part of `paver run_quality`.

To run the XSS Linter against your current branch, run the following command.

```
paver run_xsscommitlint
```

For more options for running the XSS Linter, or instructions for fixing violations, see *XSS Linter*.

5.4 Testing Open edX Features

When you create a new feature for the Open edX platform, you must write two kinds of tests for that feature: general tests that evaluate the feature on the Open edX platform, and tests that are specific to the type of feature that you create. For example, if you create a coupon codes feature for use with the edX Ecommerce service, you must write both Open edX tests and Ecommerce tests. This section describes the general tests that you must write for the Open edX platform.

5.4.1 Tests for the Open edX Platform

You must write the following types of tests for all new features that you create for the Open edX platform.

- Django tests. To learn how to test Django code, see the Testing in Django documentation provided by the Django Software Foundation.
- Acceptance tests. These tests verify behavior that relies on external systems, such as the LMS or payment processors. At a minimum, you must run these tests against a staging environment before you deploy code to production to verify that critical user workflows are functioning as expected. With the right configuration, you can also run the tests locally.

CHAPTER

SIX

ANALYTICS

The edX LMS and Studio are instrumented to enable tracking of metrics and events of interest. These data can be used for educational research, decision support, and operational monitoring.

The primary mechanism for tracking events is the *Event Tracking* API. It should be used for the vast majority of cases.

6.1 Event Tracking

The event-tracking library aims to provide a simple API for tracking point- in-time events. The event-tracking documentation summarizes the features and primary use cases of the library as well as the current and future design intent.

6.1.1 Emitting Events

Emitting from server-side python code:

```
from eventtracking import tracker
tracker.emit('some.event.name', {'foo': 'bar'})
```

Emitting from client-side JavaScript:

```
Logger.log 'some.event.name', 'foo': 'bar'
```

Note:

The client-side API currently uses a deprecated library (the `track` djangoapp) instead of the event-tracking library. Eventually, event-tracking will publish a client-side API of its own: when available, that API should be used instead of the `track`-based solution. See *Deprecated APIs*.

Naming Events

Event names are intended to be formatted as . separated strings and help processing of related events. The structure is intended to be *namespace.object.action*. The namespace is intended to be a . separated string that helps identify the source of events and prevent events with similar or identical objects and actions from being confused with one another. The object is intended to be a noun describing the object that was acted on. The action is intended to be a past tense verb describing what happened.

Examples:

- **`edx.course.enrollment.activated`**

- Namespace: `edx`
- Object: `course.enrollment`
- Action: `activated`

Choosing Events to Emit

Consider emitting events to capture user intent. These will typically be emitted on the client side when a user interacts with the user interface in some way.

Consider also emitting events when models change. Most models are not append- only and it is frequently the case that an analyst would want to see the value of a particular field at a particular moment in time. Given that many fields are overwritten, that information is lost unless an event is emitted when the model is changed.

Sensitive Information

By default, event information is written to an unencrypted file on disk. Therefore, do not include clear text passwords, credit card numbers, or other similarly sensitive information.

Size

A cursory effort to regulate the size of the event is appreciated. If an event is too large, it may be omitted from the event stream. However, do not sacrifice the clarity of an event in an attempt to minimize size. It is much more important that the event is clear.

Debugging Events

On devstack, emitted events are stored in the `/edx/var/log/tracking.log` log file. This file can be useful for validation and debugging.

Testing Event Emission

It is important to test instrumentation code since regressions can have a significant negative impact on downstream consumers of the data.

Each test can make stronger or weaker assertions about the structure and content of various fields in the event. Tests can make assertions about particular fields in the nested hierarchical structures that are events. This allows them to continue to pass even if a new global field is added to all events (for example).

Failing tests are a form of communication with future developers. A test failure is a way for you to tell those future developers that they have changed something that you did not expect to change, and that there are implications that they should think about carefully before making the change. For this reason, limit the scope of your test to details you expect to remain constant. Specifically, for eventing, this means only asserting on the presence and correctness of fields your code is adding, not the precise set of fields that happen to be present in all events today.

In general, it is acceptable for events to contain "unexpected" fields. If you add a field, most JSON parsers will accept this new field and allow the downstream code to process the event. Since that downstream code does not know about the new field it will simply be ignored.

For this reason, most of our tests do not actually make assertions about unexpected fields appearing in the events, instead they focus on the fields that they *do* expect to be present and make assertions about the values of these fields. This enables us to add global context without having to update hundreds (or even thousands) of tests that were making assertions about the exact set of fields present in the event. Instead, we prefer to only have a small number of tests

fail when making a change like this. Those tests might be making more strict assertions about the global context, for example. When a small number of targeted tests fail, they can be more effective at communicating the exact set of assumptions that were being made before that have now changed.

Assertions

The `openedx.core.lib.tests.assertions.events` module contains helper functions that can be used to make assertions about events. The key function in this module is `assert_event_matches` which allows tests to make assertions about parts of the event. The signature looks like this:

```
def assert_event_matches(expected_event, actual_event, tolerate=None):
```

The `expected_event` parameter contains the assertion that is being made. The `actual_event` parameter contains the complete event that was emitted. The `tolerate` parameter allows the test to specify the types of discrepancies that it cares about. This allows you to be very strict in assertions about some parts of the event and more lenient in other areas.

Here are examples that highlight the default settings for `tolerate`.

```
# By default, decode string values for the "event" field as JSON and compare
# the contents with the actual event. This will not raise an error.
assert_event_matches(
    {'event': {'a': 'b'}},
    {'event': '{"a": "b"}'}
)

# Ignore "unexpected" root fields. This will not raise an error even though
# the field "foo" does not appear in the expected event.
assert_event_matches(
    {'event_type': 'test'},
    {'event_type': 'test', 'foo': 'bar'}
)

# Ignore "unexpected" fields in the context. This will not raise an error
# even though the field "foo" does not appear in the expected event context.
assert_event_matches(
    {'event_type': 'test'},
    {'event_type': 'test', 'context': {'foo': 'bar'}}
)

# Overriding "tolerate" allows more strict assertions to be made.
# This assertion will raise an error!
assert_event_matches(
    {'event_type': 'test'},
    {'event_type': 'test', 'context': {'foo': 'bar'}},
    tolerate=[]
)
```

Unit testing

Test classes should inherit from `common.djangoapps.track.tests.EventTrackingTestCase`. Additionally, some helper assertion functions are available to help with making assertions about events.

Here is an example of a subclass.

```
from common.djangoapps.track.tests import EventTrackingTestCase
from openedx.core.lib.tests.assertions.events import assert_event_matches

class MyTestClass(EventTrackingTestCase):

    def setUp(self):
        # The setUp() of the superclass must be called
        super(MyTestClass, self).setUp()

    def test_event_emitted(self):
        my_function_that_emits_events()

        # If the above function only emits a single event, this can be used.
        actual_event = self.get_event()

        # This will assert that the "event_type" of the event is "foobar".
        # Note that it makes no assertions about any of the other fields
        # in the event.
        assert_event_matches({'event_type': 'foobar'}, actual_event)

    def test_no_event_emitted(self):

        my_function_that_does_not_emit()

        # This will fail if any events were emitted by the above function
        # call.
        self.assert_no_events_emitted()
```

Bok Choy Testing

Test classes should use the mixin `common.test.acceptance.tests.helpers.EventsTestMixin`. At its core, this mixin captures all events that are emitted while the test is running and allows you to make assertions about those events. Below some common patterns are outlined. By default, Bok Choy event assertions are as lenient as possible. The tests can be made more strict by passing in `tolerate=[]` to indicate that an exact match is necessary. Similarly, other flags can be passed into the `tolerate` parameter to tightly control the level of validation performed.

Wait for some events and make assertions about their content.

```
def test_foobar_event_emission(self):
    emit_foobar_event()

    # This will wait for the event to be emitted. It will time out if the
    # event is not emitted quickly enough (or not emitted at all).
    actual_events = self.wait_for_events({'event_type': 'foobar'})

    # This will compare the first event emitted with the first expected
```

(continues on next page)

(continued from previous page)

```
    # event, the second with the second etc.
    self.assert_events_match(
      [
        {'event': {'a': 'b'}}
      ],
      actual_events
    )

    # ``wait_for_events`` also accepts arbitrary callable functions to check
    # to see if an event "matches"
    def some_custom_event_filter(event):
        return event['event']['old_time'] > 10

    # This will return when some_custom_event_filter returns true for at
    # at least one event.
    actual_events = self.wait_for_events(some_custom_event_filter)

def test_multiple_events(self):
    emit_several_events()

    def my_event_filter(event):
        return event['event_type'] in ('first_event', 'second_event')

    # This will wait for 2 events to match the filter defined above. Note
    # that it makes no assertions about their ordering or content.
    actual_events = self.wait_for_events(my_event_filter, number_of_matches=2)

    # This ensures that first_event was emitted before second_event and
    # checks the payload of both events.
    self.assert_events_match(
      [
        {
          'event_type': 'first_event',
          'event': {'a': 'b'}
        },
        {
          'event_type': 'second_event',
          'event': {'a': 'other'}
        }
      ],
      actual_events
    )

def test_granular_assertion(self):

    # This foobar event is emitted first, with the "a" field set to "NOT B"
    tracker.emit('foobar', {'a': 'NOT B'})

    # A context manager can be used to ensure that the first "foobar" event
    # is ignored. It only makes assertions about the events that are emitted
    # inside this context.
    with self.assert_events_match_during(
```

(continues on next page)

(continued from previous page)

```
        {'event_type': 'foobar'},
        [
          {
            'event': {'a': 'b'}
          }
        ]
    ):
        emit_foobar_event()
```

6.1.2 Documenting Events

When you add events to the platform, your PR should describe the purpose of the event and include an example event. In addition, consider including comments that identify the purpose of the event and its fields. Your descriptions and examples can help assure that researchers and other members of the open edX community understand your intent and use the events correctly.

You might find the following references helpful as you prepare your PR.

- The *edX Platform Developer's Guide* provides guidelines for contributing to open edX.
- The edX Research Guide is a reference for information about emitted events that are included in the edX tracking logs.

6.1.3 Request Context Middleware

The platform includes a middleware class that enriches all events emitted during the processing of a given request with details about the request that greatly simplify downstream processing. This is called the `TrackMiddleware` and can be found in `edx-platform/common/djangoapps/track/middleware.py`.

6.1.4 Legacy Application Event Processor

In order to support legacy analysis applications, the platform emits standard events using `eventtracking.tracker.emit()`. However, it uses a custom event processor which modifies the event before saving it to ensure that the event can be parsed by legacy systems. Specifically, it replicates some information so that it is accessible in exactly the same way as it was before. This state is intended to be temporary until all existing legacy systems can be altered to use the new field locations.

6.2 Other Tracking Systems

The following tracking systems are currently used for specialized analytics. There is some redundancy with event-tracking that is undesirable. The event- tracking library could be extended to support some of these systems, allowing for a single API to be used while still transmitting data to each of these service providers. This would reduce discrepancies between the measurements made by the various systems and significantly clarify the instrumentation.

6.2.1 Segment

A selection of events can be transmitted to Segment in order to take advantage of a wide variety of analytics-related third party services such as Mixpanel and Chartbeat. It is enabled in the LMS if the `SEGMENT_KEY` key is set to a valid Segment API key in the `lms.yml` file. Additionally, the setting `EVENT_TRACKING_SEGMENTIO_EMIT_WHITELIST` in the `lms.yml` file can be used to specify event names that should be emitted to Segment from normal `tracker.emit()` calls. Events specified in this whitelist will be sent to both the tracking logs and Segment. Similarly, it is enabled in Studio if the `SEGMENT_KEY` key is set to a valid Segment API key in the `studio.yml` file.

6.2.2 Google Analytics

Google analytics tracks all LMS page views. It provides several useful metrics such as common referrers and search terms that users used to find the edX web site.

6.2.3 Deprecated APIs

The `track` djangoapp contains a deprecated mechanism for emitting events. Direct usage of `server_track` is deprecated and should be avoided in new code. Old calls to `server_track` should be replaced with calls to `tracker.emit()`. The celery task-based event emission and client-side event handling do not currently have a suitable alternative approach, so they continue to be supported.

CHAPTER

SEVEN

DEPLOY A NEW SERVICE

7.1 Intro

This page is a work-in-progress aimed at capturing all of the details needed to deploy a new service in the edX environment.

7.2 Considerations

7.2.1 What Does Your Service Do

Understanding how your service works and what it does helps Ops support the service in production.

7.2.2 Sizing and Resource Profile

What class of machine does your service require? What resources are most likely to be bottlenecks for your service: CPU, memory, bandwidth, something else?

7.2.3 Customers

Who will be using your service? What is the anticipated initial usage? What factors will cause usage to grow? How many users can your service support?

7.2.4 Code

What repository or repositories does your service require. Will your service be deployed from a non-public repo?

Ideally your service should follow the same release management process as the LMS. This is documented in the wiki, so please ensure you understand that process in depth.

Was the service code reviewed?

7.2.5 Settings

How does your service read in environment specific settings? Were all hard- coded references to values that should be settings, such as database URLs and credentials, message queue endpoints, and so on, found and resolved during code review?

7.2.6 License

Is the license included in the repo?

7.2.7 How does your service run

Is it HTTP based? Does it run periodically? Both?

7.2.8 Persistence

Ops will need to know the following things.

- What persistence needs does you service have
  - Will it connect to an existing database?
  - Will it connect to Mongo
- What are the least permissive permissions your service needs to do its job.

7.2.9 Logging

It is important that your application logging in built out to provide sufficient feedback for problem determination as well as ensuring that it is operating as desired. It is also important that your service log uses our deployment standards, for example, logs vs. syslog in deployment environments, and the standard log format for syslog. Can the logs be consumed by Splunk? They should not be if they contain data discussed in the Data Security section below.

7.2.10 Metrics

What are the key metrics for your application? Concurrent users? Transactions per second?

7.2.11 Messaging

Does your service need to access a message queue?

7.2.12 Email

Does your service need to send email?

7.2.13 Access to Other Service

Does your service need access to other services, either within or outside of the edX environment? Some example might be, the comment service, the LMS, YouTube, s3 buckets, and so on.

7.2.14 Service Monitoring

Your service should have a facility for remote monitoring that has the following characteristics.

- It should exercise all the components that your service requires to run successfully.
- It should be necessary and sufficient for ensuring your service is healthy.
- It should be secure.
- It should not open your service to DDOS attacks.

7.2.15 Fault Tolerance and Scalability

How can your application be deployed to ensure that it is fault tolerant and scalable?

7.2.16 Network Access

From where should your service be accessible?

7.2.17 Data Security

Will your application be storing or handling data in any of the following categories?

- Personally Identifiable Information in General, e.g., user's email addresses.
- Tracking log data
- edX confidential data

7.2.18 Testing

Has your service been load tested? What there the details of the test. What determinations can we make regarding when we will need to scale if usage trend upward? How can ops exercise your service in order to tests end-to-end integration. We love no-op-able tasks.

7.2.19 Additional Requirements

Anything else we should know about.

CHAPTER

EIGHT

WRITING GOOD CODE

8.1 edX Accessibility Guidelines

edX measures and evaluates accessibility using the World Wide Web Consortium's Web Content Accessibility Guidelines (WCAG) 2.1 (05 June 2018). All features that you merge into edX repositories are expected to conform to Level AA of this specification and to satisfy the requirements outlined in the edX Website Accessibility Policy.

The **edX Accessibility Guidelines** are intended to extend the guidance available in WCAG 2.1, with a focus on features frequently found in the Open edX platform, as well as in Learning and Content Management Systems in general.

- *Hide or expose content to targeted audiences*
- *Choose colors that meet WCAG 2.1's minimum contrast ratios*
- *Test your code for accessibility*

8.1.1 Introduction

The core mission of edX is to expand access to education for everyone. We expect any user interfaces that are developed for the Open edX platform to be usable by everyone, regardless of any physical limitations that they might have. The Open edX platform is used every day by people who might not be able to see or hear, or who might not be able to use traditional modes of computer interaction such as the mouse or keyboard.

Understanding a few core concepts about how people with disabilities use the web and web applications should give you more context for applying the guidance in this document.

- People without vision, as well as those with neurological conditions that prevent precise hand-eye coordination, cannot use a mouse. Therefore, all interactive content must be usable by keyboard alone.
- People with mobility impairments might use custom keyboards, keyboard emulators, or voice input programs.
- Some people rely on speech to interact with web applications and need to be able to address interactive elements by speaking their accessible names. Any visible labels should be a part of the control's accessible name.
- People who cannot hear or hear well require visible text captions, or an equivalent visual cue, for any audible content. This includes recorded audio and video, but also includes audible UI feedback.
- Many people with disabilities use "assistive technologies" (such as screen readers) to interact with their computers, web browsers, and web applications.
- Some of the most commonly used assistive technologies are browser plugins. Common examples include Reader Mode (for text reflow), Speech output for text when selected or on mouse hover, and thesaurus and grammar checkers for people with dyslexia.
- Learners might customize their operating system, browser, or web page display properties to make web applications easier for them to use. For example, they might increase the font size in their displays, reverse or increase contrast, or remove images.
- To allow assistive technologies to provide the maximum information to users, anything in a user interface that is conveyed visually (such as the element with focus, an element's role, its current state, and other properties) must also be conveyed programmatically. This information is often included automatically when you use native HTML5 elements, or added when you use.

Keep these core concepts in mind when you develop user interfaces, so that they can truly be used by everyone. More information is available from the W3C's Web Accessibility Initiative's publication How People with Disabilities Use the Web: Overview.

8.1.2 Accessibility Best Practices

The following sections cover some best practices and tips to keep in mind as you develop user interfaces that are WCAG 2.0 compliant.

- *Use semantic markup*
- *Make images accessible*
- *Avoid using CSS to add content*
- *Include a descriptive title attribute for all <iframe> elements*

- *Make sure form elements have labels*
- *Include link and control labels that make sense out of context*
- *Use WAI-ARIA to create accessible widgets or enhance native elements*
- *Manage the focus for pop-ups*
- *Inform users when content changes dynamically*
- *Hide or expose content to targeted audiences*
- *Choose colors that meet WCAG 2.1's minimum contrast ratios*
- *Test your code for accessibility*

8.1.3 Use semantic markup

The role, state, and associated properties of an element are exposed to users of assistive technologies either directly through the DOM (Document Object Model) or through the Accessibility API. If you use elements for purposes other than their intended purposes, you can "break" features that are designed to make web applications easier to use, resulting in confusion when expected behaviors are not available. For example, the role, state, or associated properties of an element might be incorrectly reported when you use an element in a way that it was not designed to be used, causing confusion for users who rely on assistive technologies.

If the semantics and behavior you need already exist in a native HTML5 element, you should use that element. Do not use an element because of its default style or because it provides a convenient styling hook. Here are some common examples.

Buttons

If you want a button, use the `<button>` element. Do not use a `<div>` that looks and behaves like a button.

Checkboxes

If you want a checkbox, use the `<input type=checkbox>` element. Do not try to recreate states and properties that are included with the native element, such as focus or state. If you attempt to do so, more than likely you will not fully replicate all of them. Native checkbox elements include a toggle for checked state upon `space` or `enter` keypresses, exposing its label and "checkedness" to the Accessibility API.

Headings

Use the appropriate levels of headings (`<h1>` - `h6>`) to denote a logical hierarchical order of content. Do not use headings as stylistic markup (for their physical size or appearance).

Generally there should be only one H1 on a page (though HTML5 allows restarting the heading hierarchy within Section and Article elements, it is best to avoid this).

Headings should not skip levels.

Visual hierarchy should match the semantic hierarchy.

Lists

Use ordered lists (<ol>) only when you are marking up a collection of related items whose order in the list is important. Use unordered lists (<ul>) only when you are marking up a collection of related items. Screen readers provide extra feedback and functionality for lists and other elements with semantic importance. It can be confusing or cumbersome when this feedback is inaccurately reported.

8.1.4 Make images accessible

You can make images accessible by using the alt attribute for each image, or by providing a text alternative for an image.

Text alternatives

For users who are unable to view or use non-text content (such as images, charts, applets, audio files and so on), you can provide a text alternative . A text alternative is text that non-sighted users can access in place of the non-text content.

Text alternatives must be "programmatically determinable". This means that the assistive technologies and accessibility features in browsers must be able to read and use the text.

Text alternatives must also be "programmatically associated" with the non-text content. This means that users must be able to use assistive technology to find the text alternative when they land on the non-text content.

All images require a text alternative. The only exceptions to this rule are purely decorative images or images that have text alternatives adjacent to them.

Alt attributes

Regardless of whether or not an image requires a text alternative, you must define an alt attribute for all <img> elements, even if the value of that attribute is empty (alt=""). An empty alt attribute is also called a NULL alt attribute.

If your image is purely decorative, or has a text alternative immediately adjacent to it, use a NULL alt attribute.

If an <img> element does not have a NULL alt attribute, you should make sure that the value you use in its alt attribute provides useful information to users who rely on screen readers. If an alt attribute value does not exist, screen readers will expose the path to the image as a last resort.

Best practices for non-text elements

Providing *useful* text alternatives or alt attribute values is more difficult than it sounds. Ask yourself questions about the purpose of your image to determine what would be most useful to the user.

- Is your image the only content of a link or form control?

  Your alt attribute should describe the destination of the link, or the action that will be performed. For example, a "Play" icon should have a text alternative such as "Play the 'Introduction to Linux' course video", rather than "Right-pointing triangle".

- Does your image contain text? The vast majority of images of text should include the verbatim text as the value of the alt attribute. Here are some examples of exceptions.
  - If yes, and if the same text appears adjacent to or near the image in the DOM, use a NULL value in the alt attribute, otherwise a screen reader is exposed to the same content twice.

  - If yes, and if the text within the image is there simply for visual effect (such as a skewed screenshot of computer code), use a NULL value in the `alt` attribute.

- Does your image contribute meaning to the current page or context?
  - If yes, and if the image is a simple graphic or photograph, the `alt` attribute should briefly describe the image in a way that conveys the same meaning that a sighted person would obtain from viewing the image. Context is important. A detailed description of a photograph is rarely useful to the user, unless it is in the context of a photography or art class.
  - If yes, and if the image is a graph or complex piece of information, include the information contained in the image elsewhere on the page. The `alt` attribute value should give a general description of the complex image. You can programmatically link the image with the detailed information using `aria-describedby`.

A pragmatic guide on providing useful text alternatives is included in the HTML5 specification (4.7.1.1). It provides a variety of example images and appropriate text alternatives.

A more comprehensive decision tree is available in the Web Accessibility Initiatives Images Tutorial.

8.1.5 Avoid using CSS to add content

CSS-generated content can cause many accessibility problems. Since many screen readers interact with the DOM, they are not exposed to content generated by CSS, which does not live in the DOM. There is currently no mechanism for providing alternative content for images added using CSS (either background images or pseudo elements).

Many developers think that providing screen reader-only text can be used to solve this problem. However, images added using this technique are not rendered to users who have high contrast mode enabled on their operating systems. These users are likely not using screen readers, so they cannot access the visible icon or the screen reader text.

Content injected into the DOM using JavaScript is more accessible than content added using CSS.

When adding images that represent important navigational or information elements, use `<img>` elements with appropriate `alt` attributes. For more information about making images accessible, see *Make images accessible*.

8.1.6 Include a descriptive `title` attribute for all `<iframe>` elements

Use the `title` attribute to provide a description of the embedded content to help users decide whether or not they would like to interact with this content. It is possible that `<iframe>` titles are presented out of context (such as in a list within a dialog box), so choose title text that will make sense when it is exposed out of context.

8.1.7 Include link and control labels that make sense out of context

Label text for all links and interactive controls should make sense out of context. Screen reader users have the option of listing and navigating links and form controls out of the context of the page. When a page contains vague and non-unique text such as **Click here** or **More...**, the purpose of these links is not clear without the context of surrounding text.

8.1.8 Make sure form elements have labels

All form elements must have labels, either using the label element or the aria-label or aria- labelledby attributes.

Sighted users have the benefit of visual context. It is usually quite obvious to them what the purpose is of a given form field, based on physical proximity of descriptive text or other visual cues. However, to a user with a vision impairment, who does not have the benefit of visual context, these relationships are not obvious. Users who rely on speech to interact with their computers also need a label for addressing form elements. If you correctly use the `<label>` element, text is programmatically associated with a given form element, and can then be read to the user upon focus, or used to address the form element using speech input.

Note: Screen readers often enter "forms processing mode" when they encounter a form. This mode temporarily disables all keyboard shortcuts available to users so that key presses are passed through to the control. The exception is the `TAB` key, which moves focus from one form field to the next. This means that context-sensitive help provided for form fields (such as UI help text adjacent to the form field) is not likely to be encountered by screen reader users. To remedy this situation, add an aria-describedby attribute to the input that references the help text. Doing so programmatically links the help text to the form control so that users can access it while their screen readers are in forms processing mode.

8.1.9 Use WAI-ARIA to create accessible widgets or enhance native elements

In some cases, native HTML5 elements will not provide the behavior or style options that you want. If you develop custom HTML or JavaScript widgets, make sure you add all necessary role, state, and property information for each widget, so that it can be used by users of assistive technology.

WAI-ARIA (Web Accessibility Initiative - Accessible Rich Internet Applications) is a technical specification published by the World Wide Web Consortium (W3C) that specifies how to increase the accessibility of web pages.

When you develop custom widgets, use WAI-ARIA to ensure that your custom controls are accessible, and consider the following points.

- Is the role of the widget properly identified?
- Can a user focus on and interact with your widget using the keyboard alone?
- When the state or some other property of your widget changes, are those changes conveyed using ARIA attributes to users of assistive technology?

Note: Adding an ARIA `role` overrides the native role semantics reported to the user from the Accessibility API. ARIA indirectly affects what is reported to a screen reader or other assistive technology. Adding an ARIA `role` to an element does not add the behaviors or attributes to that element. You have to do that yourself.

ARIA attributes can also be used to enhance native elements by adding helpful information specifically for users of assistive technology. Certain sectioning elements (such as `<nav>` and `<header>`) as well as generic ones (such as `<div>` with "search", "main" or "region" roles defined), receive special behaviors when encountered by assistive technology. Most screen readers announce when a user enters or leaves one of these regions, allow direct navigation to the region, and present the regions to a user in a list that they can use to browse the page out of context. Because your pages are likely to have multiple `<nav>` elements or `<div>` elements with "region" roles defined, it is important to use the `aria-label` attribute with a clear and distinct value to differentiate between them.

Example: Adding descriptive labels to HTML5 structural elements

```
<!-- the word "Navigation" is implied and should not be included in the label -->
<nav aria-label="Main">
...
</nav>

<nav aria-label="Unit">
...
</nav>

<div role="search" aria-label="Site">
...
</div>

<div role="search" aria-label="Course">
...
</div>
```

Some cautions for using WAI-ARIA

The following list outlines specific cases in which you have to be careful using WAI-ARIA.

- Setting `role="presentation"` strips away all of the semantics from a native element.
- Setting `role="application"` on an element passes all keystrokes to the browser for handling by scripts. In this case, all keyboard shortcuts provided by screen readers are disabled. You should only use `role="application"` if you can provide support for all of the application's functions via the keyboard as well as the roles, states, and properties for all of its child elements.
- Setting `aria-hidden="true"` removes an element from the Accessibility API, making it invisible to a user of assistive technology. For elements that you intend to hide from all users, setting the CSS property `display:none;` is sufficient. It is unnecessary to also set `aria-hidden="true"`. Once the content is revealed by changing the display property, it is too easy to forget to toggle the value of `aria-hidden`.

  There are legitimate use cases for `aria-hidden`, for example when you use an icon font that has accessible text immediately adjacent to it. Icon fonts can remain silent when focused on by certain screen readers, which can lead users of screen readers to suspect that they are missing important content. Icon fonts can also be rendered as nondescript glyphs by some screen readers that display what is being spoken on the screen. In these cases, it is useful to remove icon fonts using `aria- hidden`, so that screen reader users are not provided with the same information in both accessible and less-accessible formats.

Additional considerations for developing custom widgets are covered in General steps for building an accessible widget.

Specific considerations for common widgets are covered in WAI-ARIA 1.2 Authoring Practices and examples.

A quick reference list of Required and Supported ARIA attributes by role is available in the Using ARIA

8.1.10 Manage the focus for pop-ups

Do not forget to manage focus on pop-ups. Whenever a control inserts interactive content into the DOM or reveals previously hidden content (for example, pop-up menus or modal dialog boxes), you must move focus to the container. While the focus is within the menu or dialog box, keyboard focus should remain trapped within its bounds. Clicking the **Esc** key or the **Save** or **Cancel** button should close and exit the region and return focus to the element that triggered it.

Note that `<div>` and other container elements are not natively focusable. If you want to move focus to a container you must set a `tabindex="-1"` attribute for that container. You should also define a `role` and an `aria-label` or `aria-labelledby` attribute that identifies the purpose of the container.

8.1.11 Inform users when content changes dynamically

If a user action or script updates the content of a page dynamically, you should add the `aria-live="polite"` attribute to the parent element of the region that changes. Doing so ensures that the contents of the element are read to a screen reader user, even though the element does not currently have focus. This method is not intended to be used when the region contains interactive elements.

8.1.12 Hide or expose content to targeted audiences

Content that enhances the experience for one audience might be confusing or encumber a different audience. For instance, a **Close** button that looks like `X` will be read by a screen reader as the letter X, unless you hide it from the Accessibility API.

To visibly hide content that should be read by screen readers, edX makes a CSS `class="sr"` available to expose content only to non-visual users. This is achieved by displaying the content beyond the bounds of the viewport, or clipping the content to a single pixel. These techniques remove any visible trace of the element from the page, while still leaving it accessible to screen reader users. This is often referred to as displaying content "offscreen". In the following example, a visual user sees only the X, while a screen reader user hears only "Close".

```
<a href="#">
  <span aria-hidden="true">X</span>   <!-- hidden from screen reader users -->
  <span class="sr">Close</span>       <!-- exposed only to screen reader users -->
</a>
```

The choice to show or hide content for a specific audience should not be taken lightly. Extensive use of offscreen content can reduce accessibility, and is often an indicator of a user experience that relies too heavily on visual context. The following questions should help you decide whether it is appropriate to hide content from screen readers or display it offscreen.

- Would all users benefit from the content displayed offscreen?
  - If the content you are considering displaying offscreen might be useful not only for non-visual users but other users too, find a way to make the content work visually, and expose it for all users.
- Are you using only visual cues to provide important context?
  - In standard sidebar navigation, it is common practice to indicate the user's current page or section by differentiating it visually from other pages or sections in the sidebar. To visual users, it is clear that the item in the list that looks different than all the others is the page that they are currently viewing. You can make this visual context available to non-visual users with offscreen text, as demonstrated in the following example.

```
<a href="/" class="inactive">Home</a>
<a href="about/" class="active">About Us<span class="sr"> Current page</span></a>
```

Note: In the code example above, the non-breaking space prevents a screen reader from reading the text as "About UsCurrent Page".

- Does the content displayed offscreen contain any interactive elements?
  - Never include interactive elements such as links, buttons, or form inputs, in offscreen content. Doing so negatively impacts sighted keyboard-only users, who require visual focus indicators to understand what element has focus and will be the target of keyboard events.
- Are you including interactive elements in offscreen content?
  - It can be tempting to use offscreen text to improve the usability of an interactive element for non-visual users. Offscreen text is included in the Accessibility API which is used by screen readers. However, screen readers are not the only assistive technology that use the Accessibility API. Speech input software also uses the Accessibility API to identify interactive controls. In the following example, screen reader users will hear "Type your First Name", but sighted users will see only "First Name." Users who rely on speech input to interact with their computer will move focus to this element by saying "Focus on First Name" (the visual label). However, the accessible label for this element is "Type your First Name."

```
<label><span class="sr">Type your </span>First Name
  <input type="text" />
</label>
```

8.1.13 Choose colors that meet WCAG 2.1's minimum contrast ratios

A minimum contrast ratio between foreground and background colors is critical for users with impaired vision. You can check color contrast ratios using any number of tools available free online.

8.1.14 Test your code for accessibility

The only way to determine if your feature is fully accessible is to manually test it using assistive technology; however, there are a number of automated tools you can use to perform an assessment yourself. Automated tools might report false positives and might not catch every possible error, but they are a quick and easy way to detect the most common mistakes.

These are some automated tools for accessibility testing.

- Your keyboard. For information about using your keyboard to test for accessibility, see http://webaim.org/techniques/keyboard/.
- Accessibility features in Chrome's Developer Tools allow you to see the Accessibilty Tree, ARIA attributes, and computed properties
- Lighthouse auditing tools built into Chrome's Developer Tools offer automated accessibility testing.
- Accessibility Insights for Web.
- WAVE Accessibility Toolbar. This toolbar provides access to web accessibility evaluation tools that you can run in Firefox. A Chrome extension is available.

Note: By default, the Mac OSX operating system is configured to move keyboard focus to **Text boxes and lists only**. This setting also applies to browsing web pages using Safari or Firefox with a keyboard. To effectively test keyboard accessibility using a Mac, you should configure your computer to focus on **All controls**. Open **System Preferences**, and then select **Keyboard**. On the **Shortcuts** tab, check **Use keyboard navigation to move focus between controls**.

If you are a Chrome user, this behavior is controlled in a browser setting and is enabled by default. However, if you find that you cannot move focus to links while using Chrome you might need to change your browser configuration. Open **Settings**, then click **Show advanced settings**. Under **Web content**, confirm that the **Pressing Tab on a web page highlights links, as well as form fields** checkbox is selected.

To test your feature using a screen reader, you can use the following options.

- VoiceOver is a free, built-in screen reader for Mac and iOS.
- ChromeVox is a free screen reader (browser extension) for Chrome.
- NVDA is a free screen reader for Windows.
- JAWS is a screen reader for Windows. It is a commercial product but free to use in a limited-time demo mode.
- Narrator is a free, built-in screen reader for Windows.

Note: VoiceOver, NVDA, and Narrator can be configured to speak any text on screen on mouse hover.

8.2 Django Good Practices

8.2.1 Imports

Always import from the root of the project:

```
from lms.djangoapps.hologram.models import 3DExam     # GOOD
from .models import 3DExam                            # GOOD
from hologram.models import 3DExam                    # BAD!
```

The second form (relative import) only works correctly if the importing module is itself imported correctly. As long as there are no instances of the third form, everything should work. Don't forget that there are other places that mention import paths:

```
url(r'^api/3d/', include('lms.djangoapps.hologram.api_urls')),    # GOOD
url(r'^api/3d/', include('hologram.api_urls')),                   # BAD!

@patch('lms.djangoapps.hologram.models.Key', new=MockKey)         # GOOD
@patch('hologram.models.Key', new=MockKey)                        # BAD!

INSTALLED_APPS = [
    'lms.djangoapps.hologram',     # GOOD
    'hologram',                    # BAD!
]
```

CHAPTER

NINE

WRITING CODE FOR INTERNATIONALIZATION

9.1 Internationalization Coding Guidelines

Preparing code to be presented in many languages can be complex and difficult. This section presents best practices for marking English strings in source so that they can be extracted, translated, and displayed to the user in the language of their choice.

9.1.1 General Internationalization Rules

For source files to be successfully localized, you need to prepare them so that any human-readable strings can be extracted by a pre-processing step, and then have localized strings used at runtime. This preparation requires attention to detail, and unfortunately limits what you can do with strings in the code.

Follow these general rules for internationalizing your code.

1. Always mark complete sentences for translation. If you combine fragments at runtime, there is no way for the translator to construct a proper sentence in their language.
2. Do not combine strings together at runtime.
3. Limit the amount of text in strings that is not presented to the user. HTML markup is better applied after translation. If you include HTML in strings there is a chance that translators will translate your tags or attributes.
4. Use placeholders with descriptive names: `"Welcome {student_name}"` is much better than `"Welcome {0}"`.

For details, see *Style Guidelines*.

9.1.2 Editing Source Files

When you edit source files (including Python, JavaScript, or HTML template files), you should be aware of the following conventions.

1. Know what has to be at the top of the file (if anything) to prepare it for i18n.
2. Know how strings are marked for internationalization. This markup typically takes the form of a function call with the string as an argument.
3. Know how translator comments are indicated. Such comments in the file will travel with the strings to the translators, giving them context to produce the best translation. Translator comments have a `Translators:` marker and must appear on the line preceding the text they describe. In Python, multi-line comments are supported for translator comments that need to be wrapped.

The code samples below show how to do each of these things for a variety of programming languages.

Note: Take into account not just the programming language involved, but the type of file that you are preparing for internationalization. For example, JavaScript embedded in an HTML Mako template is treated differently than JavaScript in a pure .js file. There are many different escaping methods that you can use. For more details, see *Preventing Cross Site Scripting Vulnerabilities*.

Python Source Code

In most Python source code, indicate strings for translation and add translator comments as shown. For more details, refer to the Django documentation.

```
from django.utils.translation import ugettext as _

# Translators: This will help the translator
message = _("Welcome!")

# Translators: This is a very long comment that needs to wrap
# over multiple lines because it would be too long otherwise.
message = _("Hello world")
```

Some edX code cannot use Django imports. To maintain portability, XBlocks, XModules, Inputtypes and Responsetypes forbid importing Django. Each of these has its own way of accessing translations, as shown in the following examples.

```
### for XBlock & XModule:
_ = self.runtime.service(self, "i18n").ugettext
# Translators: a greeting to newly-registered students.
message = _("Welcome!")

# for InputType and ResponseType:
_ = self.capa_system.i18n.ugettext
# Translators: a greeting to newly-registered students.
message = _("Welcome!")
```

Translator comments will work in these places too, so wherever possible, provide clarifying comments for translators. However, be aware of a quirk in the Python parser. When you write translator comments, make sure the message string is on the very next line after the comment.

The following example is not correct.

```
# Translators: this comment won't be properly harvested!
message = _(
    "Long message "
    "on a few lines."
)
```

This example is correct.

```
message = _(
    # Translators: this comment will be properly harvested!
    "Long message "
    "on a few lines."
)
```

Django Template Files

In Django template files (*templates/*.html*), indicate strings for translation and add translator comments as shown.

```
{% load i18n %}

{# Translators: this will help the translator. #}
{% trans "Welcome!" %}
```

Mako Template Files

In Mako template files (*templates/*.html*), you can use all of the tools available to Python programmers. Just make sure to import the relevant functions first. Here is a Mako template example.

```
<%page expression_filter="h"/>
<%! from django.utils.translation import ugettext as _ %>
...
## Translators: message to the translator. This comment may
## wrap on to multiple lines if needed, as long as they are
## lines directly above the marked up string.
${_("Welcome!")}
```

Make sure that all Mako comments, including translators comments, begin with *two* pound signs (#).

All translated strings should be text, not HTML. This means that for display in an HTML page, the strings must be HTML-escaped. In the example above, HTML- escaping is handled through the <%page> directive with the h filter. For more information, see *Preventing Cross Site Scripting Vulnerabilities*.

To mix plain text and HTML using `format()`, you must use the `HTML()` and `Text()` functions. Use the `HTML()` function when you have a replacement string that contains HTML tags. For the `HTML()` function to work, you must use the `Text()` function to wrap the plain text translated string. Both the `HTML()` and `Text()` functions must be closed before any calls to `format()`.

```
<%page expression_filter="h"/>
<%!
from django.utils.translation import ugettext as _

from openedx.core.djangolib.markup import HTML, Text
%>
...
${Text(_("Click over to {link_start}the home page{link_end}.")).format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
)}
```

You can nest the formatting further. The rule is, any string which is HTML should be wrapped in the `HTML()` function, and any string which is not wrapped in `HTML()` should be escaped as needed to be displayed as regular text. Again, you must close the `HTML()` and `Text()` calls before making any call to `format()`.

```
<%page expression_filter="h"/>
<%!
from django.utils.translation import ugettext as _

from openedx.core.djangolib.markup import HTML, Text
%>
...
${Text(_("Click over to {link_start}the home page{link_end}.")).format(
    link_start=HTML('<a href="{}">').format(home_page_link),
    link_end=HTML('</a>'),
)}
```

For more information on proper escaping, see *Preventing Cross Site Scripting Vulnerabilities*.

JavaScript Files

To internationalize JavaScript, the HTML template (`base.html`) must first load a special JavaScript library, and Django must be configured to serve it.

```
<script type="text/javascript" src="jsi18n/"></script>
```

Then, in JavaScript files (`*.js`):

```
// Translators: this will help the translator.
var message = gettext('Welcome!');
```

For interpolation with translated strings, you must use `StringUtils.interpolate` or `HtmlUtils.interpolateHtml`, as shown in the following example.

```
var message = StringUtils.interpolate(
    gettext('You are enrolling in {courseName}'),
    {
        courseName: 'Rock & Roll 101'
    }
)
```

For more details on how to use `StringUtils` and `HtmlUtils`, see *Safe JavaScript Files*.

Note that JavaScript embedded in HTML in a Mako template file is handled differently. There, you must use the Mako syntax even within the JavaScript.

CoffeeScript Files

CoffeeScript files are compiled to JavaScript files, so you indicate strings for translation and add translator comments mostly as you would in *Javascript*.

```
`// Translators: this will help the translator.`
message = gettext('Hey there!')

# Interpolation must use JavaScript, not CoffeeScript interpolation
var message = StringUtils.interpolate(
    gettext('You are enrolling in {courseName}'),
    {
        courseName: 'Rock & Roll 101'
    }
)
```

However, because strings are extracted from the compiled .js files, some native CoffeeScript features break the extraction from the .js files. Be aware of the following rules.

1. Do not use CoffeeScript string interpolation. Doing so results in string concatenation in the .js file, preventing string extraction. Instead, use `StringUtils.interpolate` and `HtmlUtils.interpolateHtml` as documented in *Safe JavaScript Files*.
2. Do not use CoffeeScript comments for translator comments. They are not passed through to the JavaScript files.

```
# DO NOT do this:
# Translators: this won't get to the translators!
message = gettext("This won't work")

# YES do this:
`// Translators: this will get to the translators.`
message = gettext("This works")

###
Translators: This will work, but takes three lines :(
###
message = gettext("Hey there")
```

Underscore Template Files

Underscore template files are used in conjunction with JavaScript, so the same techniques that are used for localization in *Javascript* are used for Underscore template files.

Make sure that the i18n JavaScript library has already been loaded, and then use the i18n function `gettext` and the `StringUtils` function `StringUtils.interpolate` in your template, as shown in this example.

```
<%-
    StringUtils.interpolate(
        gettext('You are enrolling in {courseName}'),
        {
            courseName: 'Rock & Roll 101'
        }
    )
%>
```

Important: Due to a bug in the underlying underscore extraction library, when `StringUtils.interpolate` and `gettext` are on the same line, the library will not work properly. In such cases, the library will extract the word `gettext` rather than the actual string that needs to be extracted. Make sure to separate `StringUtils.interpolate` and `gettext` into two different lines, as shown in the example above.

Note: You must use <%- for all translated strings that do not include HTML tags, as this will HTML-escape the strings before including them in the page.

If you have a translated string that includes a mix of HTML and plain text, you must use `HtmlUtils.interpolateHtml` along with <%=. Using <%= is only acceptable when you use an `HtmlUtils` function.

```
<%=
    HtmlUtils.interpolateHtml(
        gettext('You are enrolling in {spanStart}{courseName}{spanEnd}'),
        {
            courseName: 'Rock & Roll 101',
            spanStart: HtmlUtils.HTML('<span class="course-title">'),
            spanEnd: HtmlUtils.HTML('</span>')
        }
    )
%>
```

You can access `HtmlUtils` and `StringUtils` from inside a template that is processed using `HtmlUtils.template()`. For more details regarding the use of `StringUtils` and `HtmlUtils`, see *Safe JavaScript Files*.

Currently, translator comments are not supported in underscore template files, because the underlying library does not parse them out. You should add translator comments using standard comment syntax, so that when work is done to support translator comments, the comments are already defined in your code. Additionally, translator comments in the code will enable us to answer questions from translators.

Other Types of Content

We have not yet established guidelines for internationalizing the following types of content.

- Course content (such as subtitles for videos)
- Documentation (written for Sphinx as .rst files)

9.1.3 Coverage Testing

These instructions assume that you are a developer working on internationalizing new or existing user-facing features. To test that your code is properly internationalized, you generate a set of 'dummy' translations, then view those translations on your app's page to make sure everything (scraping and serving) is working properly.

First, use the coverage tool to generate dummy files.

```
$ paver i18n_dummy
```

This step creates new dummy translations in the Esperanto directory (edx-platform/conf/local/eo/LC_MESSAGES) and the RTL directory (edx-platform/conf/local/rtl/LC_MESSAGES). DO NOT CHECK THESE FILES IN. You should discard these files once you have finished testing.

Next, restart the LMS and Studio to load in the new translation files.

```
$ paver devstack lms
$ paver devstack studio
```

Append `/update_lang/` to the root LMS or Studio URL and use the form to set the preview language. The language code `eo` can be used to specify the test language.

Instead of plain English strings, you should see specially-accented English strings that look like this example.

> Thé Fütüré øf Ønlïné Édüçätïøn # Før änýøné, änýwhéré, änýtïmé #

This dummy text is distinguished by extra accent characters. If you see plain English without these accents, it most likely means that the strings have not yet been marked for translation, or you have broken a rule. To fix this issue, follow these steps.

1. Find the strings in the source tree (either in Python, JavaScript, or HTML template code).
2. Refer to the coding guidelines above to make sure the strings have been properly externalized.
3. Rerun the scripts and confirm that the strings are now properly converted into dummy text.

This dummy text is also distinguished by "Lorem ipsum" text at the end of each string, and is always terminated with "#". The original English string is padded with about 30% more characters, to simulate languages (such as German) which tend to have longer strings than English. If you see problems with your page layout, such as columns that do not fit, or text that is truncated (the # character should always be displayed on every string), then you will probably need to fix the page layouts accordingly to accommodate the longer strings.

Finally, append `/update_lang/` to the root LMS or Studio URL and set the language code to `rtl` to view your feature in the dummy right-to-left (RTL) language. Test to make sure that the user interface is properly "flipped" to right-to-left mode. Note that certain page elements might not look correct because they are controlled by the browser. For more effective testing, switch your browser's language to Arabic or another RTL language (Hebrew, Persian, or Urdu) as well. See our RTL UI Guidelines for information about fixing any issues that you find.

When you have finished reviewing, append `/update_lang/` to the LMS or Studio URL and reset your session to your base language.

Set Preview Language

Before you set the preview language, sign in to either LMS or Studio. Then append `/update_lang/` to the root LMS or Studio URL. A form appears for you to set or clear the preview language. Set the Language code (for example use `eo` for the test language Esperanto), and then select **Submit** to set the preview language. Use the **Reset** button to reset the preview language to your default setting. Refresh your browser page to display the page in the selected language. The language persists for the duration of your session.

9.1.4 Style Guidelines

Do Not Append Strings or Interpolate Values

It can be difficult for translators to provide reasonable translations of small sentence fragments. If your code appends sentence fragments, even if it seems fine in English, the same concatenation is very unlikely to work properly for other languages.

Bad:

```
message = _("Welcome to the ") + settings.PLATFORM_NAME + _(" dashboard.")
```

In this scenario, the translator has to figure out how to translate these two separate strings. It is very difficult to translate a fragment such as "Welcome to the." In some languages, the fragments will be in a different order. For example, in Spanish this phrase would be ordered as "Welcome to the dashboard of edX".

It is much easier for a translator to figure out how to translate the entire sentence, using the pattern "Welcome to the {platform_name} dashboard."

Good:

```
message = _("Welcome to the {platform_name} dashboard.").format(platform_name=settings.
↪PLATFORM_NAME)
```

Note that you cannot concatenate (+) within the `gettext` call at all. The following example does not work.

Bad:

```
message = _(
    "Welcome to {platform_name}, the online learning platform " +
    "that hosts courses from world-class universities around the world!"
).format(platform_name=settings.PLATFORM_NAME)
```

In Python, because _() is a function, the following example works.

Good (Python only!):

```
message = _(
    "Welcome to {platform_name}, the online learning platform "
    "that hosts courses from world-class universities around the world!"
).format(platform_name=settings.PLATFORM_NAME)
```

However, in JavaScript and other languages, the `gettext` call cannot be broken up over multiple lines. You will have to live with long lines on `gettext` calls, and we make a style exception for this.

Bad:

```
message = gettext('Here is a really really long message that is' +
    'incorrectly broken over two lines.')
```

Good (JavaScript):

```
message = gettext('Here is a really really long message that is correctly left on a␣
↪single line.')
```

Use Named Placeholders

Python string formatting provides both positional and named placeholders. Use named placeholders, never use positional placeholders. Positional placeholders cannot be translated into other languages, which might need to re-order them to make syntactically correct sentences. Even with a single placeholder, a named placeholder provides more context to the translator.

Bad:

```
message = _('Today is %s %d.') % (m, d)
```

OK:

```
message = _('Today is %(month)s %(day)s.') % {'month': m, 'day': d}
```

Best:

```
message = _('Today is {month} {day}.').format(month=m, day=d)
```

Notice that in English, the month comes first, but in Spanish the day comes first. This is reflected in the .po file in the following way.

```
# fragment from edx-platform/conf/locale/es/LC_MESSAGES/django.po
msgid "Today is {month} {day}."
msgstr "Hoy es {day} de {month}."
```

The resulting output is correct in each language.

```
English output: "Today is November 26."
Spanish output: "Hoy es 26 de Noviembre."
```

Only Translate Literal Strings

As programmers, we are used to using functions in flexible ways. But translation functions such as `_()` and `gettext()` cannot be used in the same ways as other functions. At runtime, they are real functions like any other, but they also serve as markers for the string extraction process.

For string extraction to work properly, the translation functions must be called only with literal strings. If you use them with a computed value, the string extracter will not have a string to extract.

The difference between the right way and the wrong way can be very subtle, as shown in these examples.

```
# BAD: This tries to translate the result of .format()
_("Welcome, {name}".format(name=student_name))

# GOOD: Translate the literal string, then use it with .format()
_("Welcome, {name}").format(name=student_name))
```

```
# BAD: The dedent always makes the same string, but the extractor can't find it.
_(dedent("""
.. very long message ..
"""))

# GOOD: Dedent the translated string.
dedent(_("""
.. very long message ..
"""))
```

```
# BAD: The string is separated from _(), the extractor won't find it.
if hello:
    msg = "Welcome!"
else:
    msg = "Goodbye."
message = _(msg)

# GOOD: Each string is wrapped in _()
if hello:
    message = _("Welcome!")
else:
    message = _("Goodbye.")
```

Be Aware of Nested Context

When you provide strings in templated files for translation, you have to be careful of nested context. For example, consider this JavaScript fragment in a Mako template.

```
<script>
var feeling = '${_("I love you.")}';
</script>
```

When the string is rendered in French, it will produce the following invalid JavaScript.

```
<script>
var feeling = 'Je t'aime.';
</script>
```

Avoid this issue by following the best practices detailed in *Preventing Cross Site Scripting Vulnerabilities*. Here is the same example with proper escaping.

```
<%!
from django.utils.translation import ugettext as _

from openedx.core.djangolib.js_utils import js_escaped_string
%>
...
<script>
var feeling = '${_("I love you.") | n, js_escaped_string}';
</script>
```

The code with proper escaping produces the following JavaScript-escaped code.

```
<script>
var feeling = 'Je t\u0027aime.';
</script>
```

Singular vs Plural

It can be tempting to improve a message by selecting singular or plural based on a count, as shown in the following example.

```
if count == 1:
    msg = _("There is 1 file.")
else:
    msg = _("There are {file_count} files.").format(file_count=count)
```

The example above is not the correct way to choose a string, because other languages have different rules for when to use singular and when plural, and there might be more than two choices.

One option is not to use different text for different counts.

```
msg = _("Number of files: {file_count}").format(file_count=count)
```

If you want to choose based on number, you need to use another `gettext` variant to do so.

```
from django.utils.translation import ungettext
msg = ungettext("There is {file_count} file", "There are {file_count} files", count)
msg = msg.format(file_count=count)
```

The example above will properly use count to find a correct string in the translation file; you can then use that string to format in the count.

Translating Too Early

When the `_()` function is called, it will fetch a translated string using the current user's language to decide which string to fetch.

If you invoke the `_()` function before we know the user, then the wrong language might be used.

```
from django.utils.translation import ugettext as _

HELLO = _("Hello")
GOODBYE = _("Goodbye")

def get_greeting(hello):
    if hello:
        return HELLO
    else:
        return GOODBYE
```

Here the HELLO and GOODBYE constants are assigned when the module is first imported, at server startup. There is no current user at that time, so `ugettext` will use the server's default language. When we eventually use those constants to show a message to the user, they are not looked up again, and the user will get the wrong language.

There are a few ways to deal with this situation. The first is to avoid calling `_()` until we have the user.

```
def get_greeting(hello):
    if hello:
        return _("Hello")
    else:
        return _("Goodbye")
```

Another way is to use Django's `ugettext_lazy` function. Instead of returning a string, it returns a lazy object that will wait to do the lookup until it is actually used as a string.

```
from django.utils.translation import ugettext_lazy as _
```

Using this method can be tricky, because the lazy object does not act like a string in all cases.

The last way to solve the problem is to mark the string so that it will be extracted properly, but not actually do the lookup when the constant is defined.

```
from django.utils.translation import ugettext

_ = lambda text: text

HELLO = _("Hello")
GOODBYE = _("Goodbye")

def get_greeting(hello):
    if hello:
        return ugettext(HELLO)
    else:
        return ugettext(GOODBYE)
```

Here we define `_()` as a pass-through function, so the string will be found during extraction, but will not be translated too early. At runtime, we then use the real translation function to get the localized string.

Multi-line Strings

Translator notes must directly precede the string literals to which they refer. For example, the translator note here will not be passed along to translators.

```
# Translators: you will not be able to see this note because
# I do not directly prepend the line with the translated string literal.
# See the line directly below this one does not contain part of the string?
long_translated_string = _(
    "I am a long string, with many, many words. So many words that it is "
    "advisable that I be split over this line."
)
```

In such a case, make sure you format your code so that the string begins on a line directly below the translator note.

```
# Translators: you will be able to see this note.
# See how the line directly below this one contains the start of the string?
long_translated_string = _("I am a long string, with many, many words. "
                           "So many words that it is advisable that I "
                           "be split over this line.")
```

9.1.5 Additional Resources

The following links provide other resources related to internationalization.

- Django Internationalization (overview)
- Django: Internationalizing Python code
- Django Translation guidelines
- Django Format localization

9.2 Guidelines for Translating the Open edX Platform

The content of this section has moved to the new documentation site.

See https://docs.openedx.org/en/latest/translators/index.html

CHAPTER

TEN

PREVENTING CROSS SITE SCRIPTING VULNERABILITIES

10.1 Preventing Cross Site Scripting Vulnerabilities

Cross Site Scripting (XSS) vulnerabilities allow user-supplied data to be incorrectly executed as code in a web browser. It can be difficult to write code that is safe from XSS security vulnerabilities. This section presents best practices for handling proper escaping in the Open edX platform to avoid these vulnerabilities.

Note: If you become aware of security issues, do not report them in public. Instead, please email security@edx.org.

10.1.1 Philosophy and General Rules

The philosophy behind the recommendations in this section is to make things as simple as possible for developers. Protecting against XSS vulnerabilities typically requires properly escaping user-provided data before it is placed on the page. Rather than trying to determine if data is user-provided and could be compromised, we will play it safe and escape data whether it is user- provided or not. Unfortunately, because there are many different rules for escaping, you still must choose the proper type of escaping.

Here are some general rules.

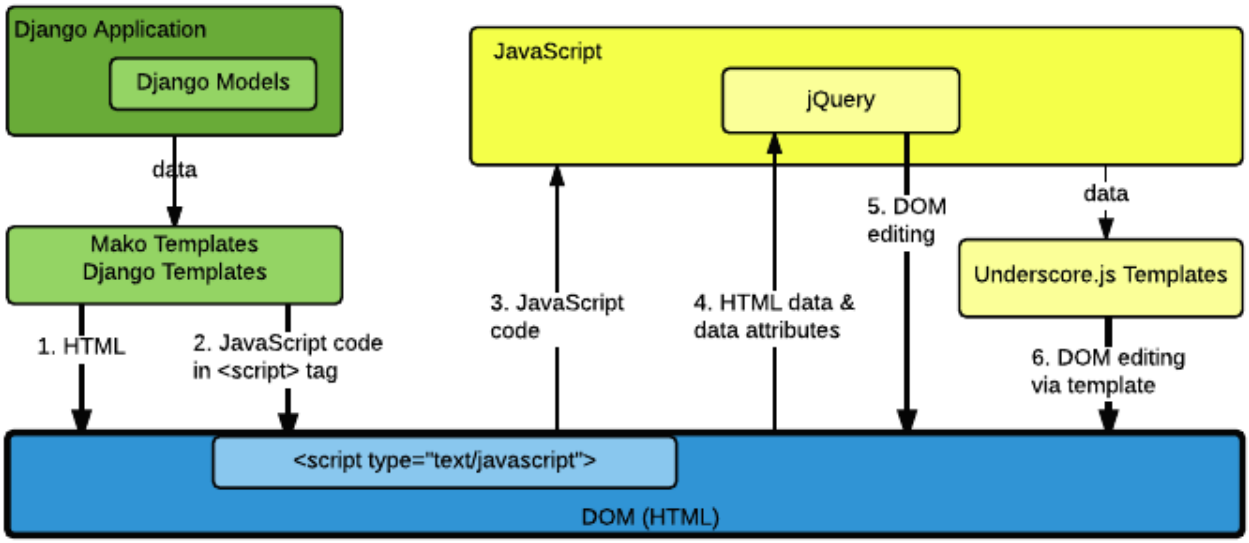
1. **Escape always.** Assume that all data is untrusted and escape it appropriately. Do not try to determine whether data could or could not be manipulated by a user.
2. **Escape late.** Delay escaping as long as possible, until you can see the actual context and understand the proper escaping that is required for the context. Browsers interpret different contexts such as HTML, URLs, CSS, and Javascript/JSON with different rules, so there are different escaping requirements based on where the data is being used in a page.

3. **Escape appropriately.** Know what kind of data you have (for example, HTML, plain text, or JSON) and where it is going (for example, HTML or JavaScript). Choose the proper escaping function based on these details.
4. **Validation is not sufficient.** Validating inputs does not replace the need to properly escape. In some cases, this may reduce the likelihood of potential problems, but proper escaping is always necessary.
5. **Do not store escaped data.** Again, because you do not know ahead of time all the places that the data will be used, you must wait until you have the proper context to decide on the proper escaping.

10.1.2 Types of Context and Escaping

Overview of Contexts

The following diagram provides a high-level overview of the relationship between the different templates, different contexts of DOM creation and manipulation, and different types of escaping. As a general rule, proper escaping is related to the context in which the data is being written, and might not match the context that will eventually be reading the data.



In the Open edX platform, data flows from the application to the initial HTML page mainly through the use of Mako templates.

Descriptions of each numbered arrow in the diagram follow.

1. This step represents the use of Mako templates to write general HTML tags (that is, tags other than `<script>` tags) to create the HTML page. Any data written to the page inside one of these HTML tags or HTML attributes must be HTML-escaped to be treated as plain text.

2. This step represents the use of Mako templates to write JavaScript inside a <script> tag in the HTML page. Data written to this context must all be JavaScript-escaped to keep it from mistakenly being treated as HTML. Note that this data should not be HTML-escaped, which must happen in a later step if it is written back to the DOM by JavaScript.
3. This step represents the loading and executing of JavaScript from the HTML page by the JavaScript engine. This step should be safe, if data was properly JavaScript-escaped earlier.
4. In this step, JavaScript might load additional data from HTML tags and attributes using the DOM. Sometimes data is passed in this way via data attributes. This data is typically read using jQuery functions, but it can be read using other JavaScript functions. The data can be in the form of plain text strings or JSON. If it is in the form of strings, this data should mostly be plain text. Be careful with the escaping in this case. Although the data is used in JavaScript, it is transmitted as HTML, and so must be HTML-escaped.
5. In this step, JavaScript is being used to edit the DOM, often by creating HTML tags or setting HTML attributes. Often this is done using jQuery functions. Since HTML tags and attributes are being written here, any plain text must be properly HTML-escaped.
6. This step represents a subset of DOM manipulation using JavaScript, specifically through use of Underscore.js templates. Although these templates have a specific syntax for escaping, because HTML tags and attributes are being written any plain text must be properly HTML-escaped.

HTML Context and Escaping

The outermost context of an HTML file is HTML. In an HTML context inside an HTML file, data is kept safe by HTML-escaping.

How HTML-Escaping Makes HTML Safe

Let's review a simple example of an XSS attack and how proper escaping might prevent such an attack. Imagine that we find the following expression in a Mako template.

```
<div>${course_name}</div>
```

Imagine further that someone uses Studio to set the course name as shown in this example, including the HTML <script> tag.

```
<script>alert('XSS attack!');</script>
```

The following resulting unsafe page source is sent to the browser.

```
<div><script>alert('XSS attack!');</script></div>
```

The browser would execute the JavaScript code in the <script>alert('XSS attack!');</script> tag. The user has injected code into the page that would display a pop-up alert, which we would not want to allow. Because this attack could contain arbitrary JavaScript that would be executed by the browser with the same trust as any JavaScript that is sent from the application, it has the potential to do something much more malicious than simply displaying a pop-up. An example might be to steal and email the user's cookies.

In Mako, you can introduce HTML-escaping for all expressions on a page using the page directive with the h filter. Here is an example of an expression that is properly HTML-escaped.

```
<%page expression_filter="h"/>
...
<div>${course_name}</div>
```

The resulting safe page source is as follows.

```
<div>&lt;script&gt;alert(&#39;XSS!&#39;);&lt;/script&gt;</div>
```

This time, the browser will not interpret the <script> tag as a JavaScript context, and instead simply displays the original string in the page.

Stripping HTML Tags

See *Preventing XSS by Stripping HTML Tags* for help with stripping HTML tags, which can be useful in cases where you want to remove certain tags rather than having them appear escaped.

JavaScript Context and Escaping

The outermost context of a JavaScript file is JavaScript. An HTML file also contains a JavaScript context inside a *<script>* tag. Inside a JavaScript context, data is kept safe by JavaScript-escaping.

How JavaScript-Escaping Makes HTML Safe

Here is an example of an expression used in a valid JavaScript context. It is created using a <script> tag inside a Mako template.

```
<script type="text/javascript">
   var courseName = "${course_name}";
   ...
</script>
```

For this example, imagine that someone uses Studio to set the course name as shown here.

```
";alert('XSS attack!');"
```

The resulting unsafe page source, sent to the browser with no escaping, would look like this.

```
<script type="text/javascript">
   var courseName = "";alert('XSS attack!');"";
   ...
</script>
```

You can see how the attacker closed out the string and again tricked the browser into executing the malicious JavaScript in the context of JavaScript. There are several reasons why you do not want to use the default HTML-escaping here.

1. JavaScript-escaping will also escape all characters that are special characters in HTML, such as <. However, JavaScript-escaping will escape < to \u003C, rather than to <. This will still keep the browser from finding an HTML tag where it does not belong.
2. The resulting string might not ultimately be used in an HTML context, so HTML entities might not be the proper escaping.

The way to properly JavaScript-escape code in Mako is shown in the following example.

```
<%! from openedx.core.djangolib.js_utils import js_escaped_string %>
...
<script type="text/javascript">
```

(continues on next page)

(continued from previous page)

```
   var courseName = "${course_name | n, js_escaped_string}";
   ...
</script>
```

The code above would produce the following safe page source.

```
<script type="text/javascript">
   var courseName = "\u0022\u003Balert(\u0027XSS attack!\u0027)\u003B\u0022\u0022\u003B";
   ...
</script>
```

CSS Context and Escaping

The browser treats any code inside a `<style>` tag or `style` attribute in an HTML page as a CSS context, or something that requires CSS parsing. CSS parsing has its own rules, and requires CSS-escaping.

In a CSS context, the following additional constraints are required to keep user supplied data safe.

- User supplied data can only appear as the value of a style property. In other words, never allow a user to supply the entire contents of the style tag or style property, or anything outside of the limited scope of an individual property value.
- User supplied URLs must use one of these safe protocols: “http:”, “https:”, or “”. Doing so prevents users from being able to supply a URL that uses the “javascript” protocol as an example.
- User supplied style property values must not contain `expression(...)` due to an IE feature that would enable arbitrary JavaScript to run.

There are no existing helper functions for these additional constraints in the platform. If you need to use user supplied data in a CSS context, you must work with edX to help expand the suite of available helpers.

For more information, see OWASP: CSS and XSS.

URL Context and Escaping

URLs require multiple types of escaping. This typically involves URL-escaping in addition to either HTML-escaping or JavaScript-escaping.

There are many special characters that are meaningful in a URL. For example, both & and = are used to designate parts of the query string. If data is being provided as a query parameter, and it might contain special characters, it must be fully URL-escaped. This is especially true with user provided data, which can contain any character. Using the JavaScript URL- escaping functions as an example, you would use the `encodeURIComponent` function on the data which will URL-escape all special characters. Here is an example.

```
var url = "http://test.com/?data=" + encodeURIComponent(userData)
```

URL-escaping is susceptible to double-escaping, meaning you must URL-escape its parts exactly once. It is best to perform the URL-escaping at the time the URL is being assembled.

Additionally, you will typically HTML-escape or JavaScript-escape a URL following the same rules for any other data added to the page, since a properly URL-escaped URL might still contain characters that are meaningful in an HTML context, such as & and `'`.

For example, when a URL is added to the `href` attribute of an anchor tag (`<a>`), it should already be properly URL-escaped, and in addition needs to be HTML-escaped at the time it is added to the HTML. When you see & between query parameters as an `&amp;` in your HTML page source, you can rest easy.

Note: If the entire URL is user provided, additional validation is required.

When an entire URL (rather that only some query parameters) is user provided, you must also validate the URL to make sure it uses a whitelisted or acceptable protocol, such as https. Doing so prevents users from being able to supply a URL that uses the "javascript" protocol as an example.

For more information, see OWASP: URL Escape.

10.1.3 Editing Template Files

When you edit template files (including Mako templates, Underscore templates, or JavaScript), use the appropriate conventions.

The topics that follow address these points for each type of file.

1. What has to be at the top of the file (if anything) to make it safe?
2. How is code properly escaped? The answer is different depending on the templating language and the context.
3. How do you properly handle internationalization and escaping together? For more information, see *Internationalization Coding Guidelines*.

Note: Remember to take into account the type of file in addition to the programming language involved. For example, JavaScript embedded in an HTML Mako template is treated differently than JavaScript in a pure .js file.

To find the proper guidelines to follow, first start with the appropriate file type below.

Django Template Files

See *Preventing XSS in Django Templates*.

Mako Template() Calls in Python Files

If a Mako template is loaded from Python outside of the general template loading scheme, the following default filters should be provided to make the template safe by default (i.e. use HTML-escaping by default).

```
template = Template(" ... ",
    default_filters=['decode.utf8', 'h'],
)
```

Mako Template Files

This topic covers the best practices for protecting Mako template files from XSS vulnerabilities.

To convert a legacy Mako template to be safe by default, it is recommended that you complete the following steps.

1. Read through the subtopics in this section and become familiar with the current best practices.
2. Follow the step-by-step instructions detailed in *Making Legacy Mako Templates Safe by Default*, which will often refer back to this section.

HTML-Escape by Default in Mako

For Mako templates, all expressions use HTML-escaping by default. This is accomplished by adding the following directive to the very top of each template.

```
<%page expression_filter="h"/>
```

Using this default HTML-escaping, the following combination represents an HTML-escaped expression.

```
<%page expression_filter="h"/>
...
${data}
```

Note: Mako templates can only have a single <%page> tag. If there is already a <%page> used for args, you must combine the two.

If you need to disable the default filters, you must use the `n` filter as the first filter. This can be seen in some of the examples below.

For a more in depth understanding of `n` filters, see *Mako Filter Ordering and the n Filter*.

Determining the Context in Mako

Most of the Mako template files are in an HTML context. That is why HTML-escaping is a good default option.

A JavaScript context is often setup implicitly through the use of the <%static:require_module> tag. In our legacy code, you might also see explicit <script> or <script type="text/javascript"> tags that initiate a JavaScript context. There are some exceptions where a <script> tag uses a different type that should be treated as an HTML context rather than a JavaScript context, for example, in <script type="text/template">.

Also, make sure you follow the best practices for *URL Context and Escaping* when working with URLs, and *CSS Context and Escaping* when in the context of a <style> tag or style attribute.

HTML Context in Mako

Most Mako expressions in an HTML context will already be properly HTML-escaped. See *HTML-Escape by Default in Mako*.

The default HTML-escaping is all that is required, even when passing JSON to a data attribute that might later be read by JavaScript. See the following example.

```
<%page expression_filter="h"/>
...
<div
    data-course-name='${course.name}'
    data-course-options='${json.dumps(course.options)}'
></div>
```

For translations that contain no HTML tags, the default HTML-escaping is enough. You must only import and use ugettext as shown in the following simple example.

```
<%page expression_filter="h"/>
<%!
from django.utils.translation import ugettext as _
%>
...
${_("Course Outline")}
```

For more complicated examples of translations that mix plain text and HTML, use the HTML(), Text(), and format() functions. Use the HTML() function when you have a replacement string that contains HTML tags. For the HTML() function to work, you must first use the Text() function to wrap the plain text translated string. Both the HTML() and Text() functions must be closed before any calls to format(). You will not use the Text function where you don't need the HTML() function. See the following example for how to import and use these functions.

```
<%page expression_filter="h"/>
<%!
from django.utils.translation import ugettext as _

from openedx.core.djangolib.markup import HTML, Text
%>
...
${Text(_("Click over to {link_start}the home page{link_end}.")).format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
)}
```

For a deeper understanding of why you must use `Text()` when using `HTML()`, see *Why Do I Need Text() with HTML()?*.

For more details about translating strings and ensuring proper escaping, see *Internationalization Coding Guidelines*.

There are times where a block of HTML is retrieved using a function in a Mako expression, such as in the following example.

```
<%page expression_filter="h"/>
from openedx.core.djangolib.markup import HTML
...
${HTML(get_course_date_summary(course, user))}
```

In this example, you use the `HTML()` function to declare the results of the function as HTML and turn off the default HTML-escaping. Using the `HTML()` function by itself can be very dangerous, unless you make sure that the function returning the HTML has itself properly escaped any plain text.

JavaScript Context in Mako

As a general guideline, JavaScript in Mako templates should be kept to an absolute minimum for a number of reasons.

- It is very difficult to mix syntax appropriately, which can lead to bugs, some of which might lead to security issues.
- The JavaScript code cannot easily be tested.
- The JavaScript code does not get included for code coverage.

For new code, the only JavaScript code in Mako that is appropriate is the minimal RequireJS code used to glue the server side and client side code. Often this is done with factory setup code to pass data to the client.

Special Mako filters are required for working with Mako expressions inside a JavaScript context.

When you need to dump JSON in the context of JavaScript, you must use either the `js_escaped_string` or `dump_js_escaped_json` filters.

With `js_escaped_string` you must supply the enclosing quotes. When `None` is supplied to `js_escaped_string`, it results in an empty string for convenience.

Often, the JavaScript context is set up implicitly through the use of `<%static:require_module>`. In our legacy code, you might also see explicit `<script>` or `<script type="text/javascript">` tags initiating a JavaScript context.

Here is an example of how to import and use `js_escaped_string` and `dump_js_escaped_json` in the context of JavaScript in a Mako template.

```
<%namespace name='static' file='static_content.html'/>
<%!
from openedx.core.djangolib.js_utils import (
    dump_js_escaped_json, js_escaped_string
)
%>
...
<%static:require_module module_name="js/course_factory" class_name="CourseFactory">
    CourseFactory({
        course_name: '${course.name | n, js_escaped_string}',
        course_options: ${course.options | n, dump_js_escaped_json},
        course_max_students: ${course.max_students | n, dump_js_escaped_json},
        course_is_great: ${course.is_great | n, dump_js_escaped_json},
```

(continues on next page)

(continued from previous page)

```
    });
</%static:require_module>
```

If you have a string that already contains JSON rather than a Python object, see *Strings Containing JSON in Mako* for how to resolve this situation.

In general, the JavaScript code inside a Mako template file should be succinct, simply providing a bridge to a JavaScript file. For legacy code with more complicated JavaScript code, you should additionally follow the best practices documented for *JavaScript Files*.

URL Context in Mako

To properly URL-escape in Python, you can use the urllib package.

For more details about URLs, see *URL Context and Escaping*.

Mako Defs

In a Mako `%def` we encounter one of the rare cases where we need to turn off default HTML-escaping using `| n, decode.utf8`. In the example below, this is done because the expression assumes that the required JavaScript-escaping was already performed by the caller.

Be extremely careful when you use `| n, decode.utf8`, and make sure the originating code is properly escaped. Note that the `n` filter turns off all default filters, including the default `decode.utf8` filter, so it is added back explicitly. Here is an example.

```
<%page expression_filter="h"/>
...
<%def name="require_module(module_name, class_name)">
    <script type="text/javascript">
        ...
        ${caller.body() | n, decode.utf8}
        ...
    </script>
</%def>
```

For more information, see Mako: Defs and Blocks.

JavaScript Files

React (JSX) files

New front-end code should use React, which is detailed in *Preventing XSS in React*.

Legacy JavaScript files

Note: The following instructions detail legacy technologies that have been deprecated, but are still actively used in edx-platform. For new code, see *React (JSX) files*.

JavaScript files are often used to perform DOM manipulation, and must properly HTML-escape text before inserting it into the DOM.

The UI Toolkit introduces various `StringUtils` and `HtmlUtils` that are useful for handling escaping in JavaScript. You can declare `StringUtils` and `HtmlUtils` as dependencies using RequireJS `define`, as seen in the following example.

```
define(['backbone',
        'underscore',
        'gettext',
        'edx-ui-toolkit/js/utils/string-utils',
        'edx-ui-toolkit/js/utils/html-utils'],
    function (Backbone, _, gettext, StringUtils, HtmlUtils) {
        ...
```

If you are working with code that does not use RequireJS, then this approach will not be possible. In this situation you can access these functions from the global `edx` namespace instead. For more information, see *JavaScript edx Namespace*.

The following `HtmlUtils` functions all make use of `HtmlUtils.HtmlSnippet`. An HTML snippet is used to communicate to other functions that the string it represents contains HTML that has been safely escaped as necessary.

The `HtmlUtils.ensureHtml()` function will ensure you have properly escaped HTML by HTML-escaping any plain text string, or simply returning any HTML snippet provided to it.

If you must perform string interpolation and translation, and your string does not contain any HTML, then use the plain text `StringUtils.interpolate()` function as follows. This function will not escape, and follows the best practice of delaying escaping as late as possible. Since the result is a plain text string, it would properly be treated as unescaped text by any of the `HtmlUtils` functions.

```
StringUtils.interpolate(
    gettext('You are enrolling in {courseName}'),
    {
        courseName: 'Rock & Roll 101'
    }
);
```

If you are performing string interpolation and translation with a mix of plain text and HTML, then you must perform HTML-escaping early and the result can be represented by an HTML snippet. Use the `HtmlUtils.HTML()` function to wrap any string that is already HTML and must not be HTML-escaped. The function `HtmlUtils.interpolateHtml()` will perform the interpolations and will HTML-escape any plain text and not HTML-escape anything wrapped with `HtmlUtils.HTML()`. See the following example.

```
HtmlUtils.interpolateHtml(
    gettext('You are enrolling in {spanStart}{courseName}{spanEnd}'),
    {
        courseName: 'Rock & Roll 101',
        spanStart: HtmlUtils.HTML('<span class="course-title">'),
        spanEnd: HtmlUtils.HTML('</span>')
```

(continues on next page)

(continued from previous page)

```
    }
);
```

You can also use `HtmlUtils.joinHtml()` to join together a mix of HTML snippets and plain text strings into a larger HTML snippet where each part will be properly HTML-escaped as necessary. See the following example.

```
HtmlUtils.joinHtml(
    HtmlUtils.HTML('<p>'),
    gettext('This is the best course.'),
    HtmlUtils.HTML('</p>')
)
```

Often, much of the preparation of HTML in JavaScript can be written using an Underscore.js template. The function `HtmlUtils.template()` provides some enhancements for escaping. First, it makes `HtmlUtils` available inside the template automatically. Also, it returns an HTML snippet so that other `HtmlUtils` functions know not to HTML-escape its results. It is assumed that any HTML-escaping required will take place inside the Underscore.js template. Follow the best practices detailed in *Underscore.js Template Files*.

The final step of DOM manipulation in JavaScript often happens using JQuery. There are some JQuery functions such as `$.text()`, `$.attr()` and `$.val()` that expect plain text strings and take care of HTML-escaping its input for you.

There are other JQuery functions such as `$.html()`, `$.append()` and `$.prepend()` that expect HTML and add it into the DOM. However, these functions do not know whether or not they are being provided properly escaped HTML as represented by an HTML snippet.

If you are working with a Backbone.js element, as represented by `el` or `$el`, you can use the JQuery methods directly, as in the following example.

```
this.parentElement.append(this.$el);
```

However, if you are creating the element through one of the other `HtmlUtils` functions, you must use `HtmlUtils.setHtml()`, `HtmlUtils.append()` and `HtmlUtils.prepend()` in place of the JQuery equivalents. These `HtmlUtils` JQuery wrappers respect HTML snippets, and can be used as seen in the following example.

```
HtmlUtils.setHtml(
    this.$el,
    HtmlUtils.joinHtml(
        HtmlUtils.HTML('<p>'),
        gettext('This is the best course.'),
        HtmlUtils.HTML('</p>')
    )
);
```

In the case of Backbone.js models, although attributes can be retrieved using the `get()` or `escape()` functions, you should avoid using the `escape()` function, which will HTML-escape the retrieved value. It is preferable to use the `get()` function and delay escaping until the time of rendering, which is often handled using an Underscore.js template.

To properly URL-escape, you can use the JavaScript functions `encodeURI` and `encodeURIComponent`. The following example shows how to properly URL-escape user provided data before it is used as a query parameter.

```
var url = "http://test.com/?data=" + encodeURIComponent(userData)
```

For more information about URLs, see *URL Context and Escaping*.

JavaScript `edx` Namespace

If you are working with code that does not use RequireJS, then it is not possible to import the `StringUtils` and `HtmlUtils` functions in the regular way. In this situation you can access these functions instead from the global `edx` namespace, as follows:

```
edx.StringUtils.interpolate(...);
edx.HtmlUtils.setHtml(...);
```

CoffeeScript Files

For CoffeeScript files, follow the same guidelines as provided for *JavaScript files*, but using the CoffeeScript syntax.

Underscore.js Template Files

The best way to HTML-escape expressions in an Underscore.js template is to use the <%- tag, which will perform the HTML-escaping.

There are some exceptions where you must use a combination of <%=, which does not escape, and one of the UI Toolkit `HtmlUtils` functions. One example is when you use the `HtmlUtils.interpolateHtml()` function to translate strings that consist of a mix of plain text and HTML. You can easily gain access to the `HtmlUtils` object inside a template by rendering the Underscore.js template using the `HtmlUtils.template()` function.

If you need to pass an HTML snippet to a template, which has already been HTML-escaped, you should name the variable with an `Html` suffix, and use `HtmlUtils.ensureHtml()` to ensure that it was in fact properly HTML- escaped. See the following example.

```
<%= HtmlUtils.ensureHtml(nameHtml) %>
```

For more details about using the `HtmlUtils` utility functions, see *JavaScript Files*.

10.1.4 Making Legacy Mako Templates Safe by Default

This topic provides a step-by-step set of instructions for making our Mako templates safe by default. For best practices to use when you write a new Mako template, see *Mako Template Files*.

By default, our Mako templates perform no escaping for expressions. We refer to this as not being "safe by default". Our intention is get to the state where our Mako templates *are* "safe by default", by ensuring that Mako template expressions perform HTML-escaping by default.

Note: It is important to understand that HTML-escaping might not be the right thing to do in all cases, but it is a good starting place. Additional escaping filters are available to help with other scenarios.

Due to valid exceptions to the general rule of HTML-escaping, it is not possible to configure escaping for all Mako templates in the entire platform without introducing errors.

The current process is for developers to make changes to each Mako template to ensure that all expressions use HTML-escaping by default. For details, see *Set HTML-Escaping Filter as Default*.

The following topics describe the steps you need to take to make your Mako templates safe by default. Although we have attempted to cover as many scenarios as possible, we are sure to have missed some cases. If you are unsure about what to do, reach out and ask for help. For contact information, see the Getting Help page on the Open edX portal .

Note: If you come across an old template that is no longer in use and can be cleaned out of the platform, help to remove the template rather than following these steps.

Set HTML-Escaping Filter as Default

Start by adding the following line to the very top of your Mako template.

```
<%page expression_filter="h"/>
```

It is important to understand that this change will affect all expressions in your Mako template. Although HTML-escaping is a reasonable default, it might cause issues for certain expressions, including HTML that cannot be escaped.

Also, be careful not to have multiple <%page> tags in a Mako template.

Run the XSS Linter

After setting HTML-escaping by default for the Mako template, run the XSS Linter with the following command.

```
./scripts/xsslint/xss_linter.py
```

Accuracy and completeness of the linter are not guaranteed, so test your work after fixing all violations.

For more detailed instructions on using the linter, see *XSS Linter*.

Fix Downstream JavaScript and Underscore.js Templates

Because Mako templates only generate the initial page source, you should ensure that any downstream JavaScript files or Underscore.js templates also follow the best practices.

When you have found the proper downstream JavaScript and Underscore.js template files, you can again run the *XSS Linter* on these files.

For help navigating our client side code, see Navigating JavaScript and Underscore.js Templates

10.1.5 XSS Linter

The XSS linter is a tool to help you make sure that you are following best practices inside edx-platform. It is not yet possible to run the linter against other repositories.

To run the linter on the changes in your current Git branch, use the following command.

```
paver run_xsscommitlint
```

To run the linter on the entire edx-platform repository, use the following command.

```
./scripts/xsslint/xss_linter.py
```

You can also lint an individual file or recursively lint a directory. Here is an example of how to lint a single file.

```
./scripts/xsslint/xss_linter.py cms/templates/base.html
```

For additional options that you can use to run the linter, use the following command.

```
./scripts/xsslint/xss_linter.py --help
```

The following code block shows sample output from the linter.

```
lms/templates/courseware/courseware-error.html: 17:7: mako-wrap-html:       ${_('There␣
↪has been an error on the {platform_name} servers').format(
lms/templates/courseware/courseware-error.html: 18:1:                           platform_
↪name=u'<span class="edx">{}</span>'.format(settings.PLATFORM_NAME)
lms/templates/courseware/courseware-error.html: 19:1:                          )}
```

Each line of linter output has the following parts.

1. The path of the file containing the violation.
2. The line number and column, for example `17:7` above, where the violation begins. In the case of Mako expressions, this will be the start of the entire expression.
3. A violation ID such as `mako-wrap-html` that represents the particular type of violation. This only appears on the first line of the violation. Additional lines may appear for context only. For more details on individual violations, run the linter with `--help`, or see *Linter Violations*.
4. The full line of code found at the provided line number.

This linter is relatively new, so if you see excessive false positives, such as a directory that should possibly be skipped, please provide feedback. The same is true if you spot an issue that was not caught by the linter. You can reach us using the Getting Help page on the Open edX portal.

Disabling Violations

If you need to disable a violation, add the following disable pragma to a comment at the start of the line before the violation, or at the end of the first line of the violation. Use the comment syntax appropriate to the file you are editing.

Here is example syntax for a Mako template.

```
## xss-lint: disable=mako-invalid-js-filter,mako-js-string-missing-quotes
```

Here is example syntax for an Underscore.js template.

```
<% // xss-lint: disable=underscore-not-escaped %>
```

Linter Violations

The following topics explain the meaning of each violation ID and what you must do to resolve each violation.

- *python-requires-html-or-text*
- *python-wrap-html*
- *underscore-not-escaped*

javascript-concat-html

Do not use + concatenation on strings that contain HTML. Instead, use `HtmlUtils.interpolateHtml()` or `HtmlUtils.joinHtml()`. For more details on proper use of `HtmlUtils`, see *JavaScript Files*.

javascript-escape

Avoid calls to `escape()`, especially in Backbone.js. Instead, use the Backbone.js model `get()` function, and perform the escaping in the template. You can also use `HtmlUtils` functions, or JQuery's `text()` function for proper escaping. For more details, see *JavaScript Files*.

javascript-interpolate

For interpolation in JavaScript, use `StringUtils.interpolate()` or `HtmlUtils.interpolateHtml()` as appropriate. For more details, see *JavaScript Files*.

javascript-jquery-append

Do not use JQuery's `append()` with an argument that might contain unsafe HTML. The linter allows a limited number of ways of coding with `append()` that it considers safe. Each of these safe techniques are detailed below.

Here is some example code with a violation.

```
// Do NOT do this
self.$el.append(
    _.template(teamActionsTemplate)({message: message})
);
```

One way to make this code safe is by replacing the `append()` call with a call to `HtmlUtils.append()`, as seen in this example.

```
// DO this
HtmlUtils.append(
    self.$el,
    HtmlUtils.template(teamActionsTemplate)({message: message})
);
```

Another way to make this code safe is to continue to use JQuery's `append()`, but to pass as an argument to `append()` the result of calling `toString()` on any `HtmlUtils` call, as in the following example.

```
// DO this
self.$el.append(
    HtmlUtils.template(teamActionsTemplate)({message: message}).toString()
);
```

You can also use JQuery `append()` with variables that represent an element, as designated by starting with a `$` or ending in `El`, such as `$element` or `sampleEl`. You can also use the `$el` element from Backbone.js.

Here is an example with one of the above mentioned safe variables.

```
// DO this
self.$el.append($button);
```

For more details regarding `HtmlUtils`, see *JavaScript Files*.

javascript-jquery-html

In some cases, JQuery's `html()` function is used with a string that does not contain any HTML tags. If this is the case, just use JQuery`s `text()` function instead. Otherwise, you can replace the `html()` call with a call to `HtmlUtils.setHtml()`, or you can call `toString()` on any `HtmlUtils` function inside the `html()` call.

For more detailed examples, see *javascript-jquery-append*.

javascript-jquery-insert-into-target

JQuery DOM insertion calls that take a target, for example `appendTo()`, can only be called from element variables. For example, you could use `$el.appendTo()`, but you cannot use `anyOldVariable.appendTo()`.

Alternatively, you could refactor to use a different JQuery method, including alternatives available in `HtmlUtils`.

For more details on legal names for element variables, see *javascript-jquery-append*.

javascript-jquery-insertion

JQuery DOM insertion calls that take content and do not have an `HtmlUtils` equivalent, for example `before()`, must use other `HtmlUtils` calls to be safe. One option is to refactor your code to use `HtmlUtils.append()`, `HtmlUtils.prepend()`, or `HtmlUtils.setHtml()`. Another alternative is to use `toString()` whenever you use an `HtmlUtils` call.

For example, let us look at the following JQuery `after()` call that is considered unsafe.

```
// Do NOT do this
this.button.after(message);
```

Instead, you could refactor the above code to create `message` using `HtmlUtils`, and then complete the refactor using `HtmlUtils.ensureHtml()`, as seen in the following example.

```
// DO this
messageHtml = HtmlUtils.template(messageTemplate)({message: message});
this.button.after(
    HtmlUtils.ensureHtml(messageHtml).toString()
);
```

javascript-jquery-prepend

Do not use JQuery's `prepend()` with an argument that might contain unsafe HTML. The linter allows a limited number of ways of coding with `prepend()` that it considers safe. For details of these safe techniques, see those described for *javascript-jquery-append*.

mako-html-entities

Once a Mako template is marked safe by default, HTML entities such as `&` should instead be plain text such as & because they will be escaped with the rest of the expression. If the entity appears in the midst of HTML, it should probably be wrapped with a call to `HTML()`.

Here is a violation as an example.

```
## Do NOT do this
<%page expression_filter="h"/>
...
${_("Details &amp; Schedule")}
```

Here is the corrected code.

```
## DO this
<%page expression_filter="h"/>
...
${_("Details & Schedule")}
```

mako-invalid-html-filter

The only valid alternative to the default HTML filter when a template is marked safe by default, is to disable HTML-escaping in one of the following ways.

```
## DO this sparingly
${HTML(x)}
## or
${x | n, decode.utf8}
```

Important: Use these functions only in the rare cases where you already have properly escaped safe HTML, and you cannot move the HTML generation to the template.

If you must disable HTML-escaping, of the two alternatives above, using `HTML()` is preferred, unless the context is ambiguous and `HTML()` does not make sense, such as in certain Mako defs.

mako-invalid-js-filter

There is a limited set of filters that the linter considers safe in a JavaScript context, so you must use one of the following safe filters.

```
<%!
from openedx.core.djangolib.js_utils import dump_js_escaped_json
%>
...
${x | n, dump_js_escaped_json}

## or
<%!
from openedx.core.djangolib.js_utils import js_escaped_string
%>
...
${x | n, js_escaped_string}

## or DO this sparingly
${x | n, decode.utf8}
```

Important: Only in the rare case where you already have properly JavaScript-escaped safe HTML, and you cannot move the JavaScript to a JavaScript file or Underscore.js template, can you use the filter `| n, decode.utf8`. This filter turns off all escaping.

Take note of any expression that was mistakenly using `| h` in a JavaScript context. Since the data inside the expression, `x` in the above example, will no longer be HTML-escaped in Mako when you remove the `| h` filter, pay extra attention to ensuring that HTML-escaping is being performed in the downstream JavaScript.

For help using these filters, see *JavaScript Context in Mako*.

mako-js-html-string

Do not embed Mako expressions directly into a JavaScript string that uses HTML. JavaScript in a Mako template should be just enough to pass variables from Mako to JavaScript. Anything more complicated is likely to cause escaping issues.

Here is a sample violation.

```
// Do NOT do this
var invalid = '<strong>${x | n, js_escaped_string}</strong>'
```

Instead, simplify the data passing from Mako to JavaScript as follows.

```
// DO this
var valid = '${x | n, js_escaped_string}'
```

You can then use the above `valid` variable using any of the JavaScript `HtmlUtils` functions, or in an Underscore.js template.

mako-js-missing-quotes

A Mako expression using the `js_escaped_string` filter must be wrapped in quotes.

```
// Do NOT do this
var message = ${msg | n, js_escaped_string}

// DO this
var message = '${msg | n, js_escaped_string}'
```

mako-missing-default

The Mako template is missing the directive that makes it safe by default. Add the following to the top of the Mako template file.

```
<%page expression_filter="h"/>
```

It is important to understand that this will add HTML-escaping to all Mako expressions in the template. The linter may report additional problems once this has been done, so you will want to run it again after this is in place.

mako-multiple-page-tags

A Mako template can only have one <%page> tag. If the template already uses this tag to pass arguments and you mistakenly add a second <%page> tag to make the template safe by default, you must combine these two tags to resolve the violation.

The following example violates this rule.

```
## Do NOT do this
<%page expression_filter="h" />
...
<%page args="section_data" />
```

Here is the corrected code.

```
## DO this
<%page args="section_data" expression_filter="h" />
```

mako-unknown-context

The linter could not determine if the context is JavaScript or HTML. In addition to using the disable pragma detailed in *Disabling Violations*, please report the issue through the Getting Help page on the Open edX portal.

mako-unparseable-expression

This violation likely means that there is a syntax error in the Mako template. If the template is valid, in addition to using the disable pragma detailed in *Disabling Violations*, please report the issue through the Getting Help page on the Open edX portal.

mako-unwanted-html-filter

Once the page level directive has been added to make the Mako template safe by default, any use of the `h` filter in an expression is redundant. These `h` filters should be removed.

django-trans-missing-escape

Each translation string needs to be properly escaped before being shown to the user. This error is raised when a `trans` tag has a missing escape filter. Translation strings should be passed through django's builtin `force_escape` filter.

```
## DO this
{% trans 'Log out' as tmsg %}{{tmsg|force_escape}}
```

The `force_escape` filter should be on the same line as the `trans` expression and should not be moved to next line.

```
## DO NOT do this
{% trans 'Log out' as tmsg %}
{{tmsg|force_escape}}
```

django-trans-invalid-escape-filter

This error is raised when the `trans` tag is escaped via unknown filter.

```
## DO Not do this
{% trans 'Log out' as tmsg %}{{tmsg|some_unknown_filter}}
```

django-trans-escape-variable-mismatch

This error is raised when there is a mismatch between the `trans` tag expression variable and the filter expression variable.

```
## DO Not do this
{% trans 'Log out' as tmsg %}{{some_other_variable|force_escape}}
```

django-html-interpolation-missing

Whenever there are some html tags that need to be used in the translation string and we do not want them to be escaped then we have to interpolate such html via a custom `interpolate_html` tag.

This error is raised when a trans/blocktrans tag includes html directly, rather than using the `interpolation_html` tag.

In the case of a `trans` tag:

```
## DO NOT do this
{% trans "some text <a href='some-path'>link text to display</a>." %}
```

The correct way is to use a variable to store the translation string and use the `interpolate_html` tag to escape that variable like below.

```
## DO this
{% trans "Some text {start_link}link text to display{end_link}." as tmsg %}
{% interpolate_html tmsg start_link='<a href='some-path'>'|safe end_link='</a>'|safe %}
```

Important: Add `{% load django_markup %}` to the top of the file to be able to use `interpolate_html` in the file.

In the case of a `blocktrans` tag:

```
## DO NOT do this
{% blocktrans%}
some text <a href="some-path">link text to display</a>.
{% endblocktrans %}
```

Instead we should use the `interpolate_html` tag to HTML-escape the string, except for parts that are real HTML that are marked as `safe`.

```
## DO this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text to display{end_link}.
{% endblocktrans %}
{% interpolate_html msg start_link='<a href="'|add:some_url|add:'">'|safe end_link='</a>
↪'|safe %}
```

Important: One thing to keep in mind while using `interpolate_html` is to correctly map the `trans/blocktrans` tag string output variable to the `interpolate_html` tag. For example, in the above case the `msg` variable used in `interpolate_html` must match the `trans/blocktrans` tag output variable (also `msg`), otherwise the linter will complain.

---

django-html-interpolation-missing-safe-filter

This error is raised when the injected html is not marked as safe via a `safe` filter. Not marking it safe would cause the `interpolate_html` tag to not only HTML-escape the translation string, but it would also HTML-escape the true HTML arguments that were passed to it.

```
## DO NOT do this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text to display{end_link}.
{% endblocktrans %}
{% interpolate_html msg start_link='<ahref="'|add:some_url|add:'">' end_link='</a>'%}
```

This linter violation is also raised when some unknown filter is being used instead of `safe`.

```
## DO NOT do this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text for to display{end_link}.
{% endblocktrans %}
{% interpolate_html msg start_link='<ahref="'|add:some_url|add:'">'|unknown_filter end_
↪link='</a>'|safe%}
```

Instead, the `**kwargs` sent to `interpolate_html` must be marked safe.

```
## DO this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text to display{end_link}.
{% endblocktrans %}
{% interpolate_html msg start_link='<ahref="'|add:some_url|add:'">'|safe end_link='</a>
↪'|safe %}
```

django-html-interpolation-invalid-tag

This violation is raised when `interpolate_tag` has missing arguments.

```
## DO NOT do this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text to display{end_link}.
{% endblocktrans %}
{% interpolate_html %}
```

`interpolate_html` requires one argument and a set of keyword arguments. The first positional argument is the output variable of the `trans/blocktrans` tag, which is the translation string that needs to be HTML-escaped. The second set of arguments are keyword arguments that are injected into the translation string.

```
## DO this
{% blocktrans trimmed asvar msg %}
some text {start_link}link text to display{end_link}.
{% endblocktrans %}
{% interpolate_html msg start_link='<ahref="'|add:some_url|add:'">'|safe end_link='</a>
↪'|safe %}
```

django-blocktrans-missing-escape-filter

This error is raised when a `blocktrans` tag is not nested under an escape filter expression.

```
## DO NOT do this
{% blocktrans %}
    You have been enrolled in {{ course_name }}
{% endblocktrans %}
```

Instead do this

```
## DO this
{% filter force_escape %}
{% blocktrans %}
    You have been enrolled in {{ course_name }}
{% endblocktrans %}
{% endfilter %}
```

django-blocktrans-parse-error

This error is raised when the `blocktrans` closing tag is missing.

```
## DO NOT do this
{% filter force_escape %}
{% blocktrans %}
Some translation string
{% endfilter %}
```

django-blocktrans-escape-filter-parse-error

This error is raised when there is a parsing error in the filter expression. Mostly this should be found by the django template parser. For example, in the code snippet below, the filter `force_escape` expression is not properly closed.

```
## DO NOT do this
{% filter force_escape
{% blocktrans %}
Some translation string
{% endblocktrans %}
{% endfilter %}
```

python-close-before-format

You must close any call to `Text()` or `HTML()` before calling `format()`. Another way to state this is that you should only pass a single literal string to `Text()` or `HTML()`.

Here is an example of this violation. Note that the problem is subtle, and that there is only a single `)` before the call to `format()`, closing the `_()` call, but not the `Text()` call.

```
## Do NOT do this
${Text(_("Click over to {link_start}the home page{link_end}.").format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
))}
```

Here is a corrected version of the same code block.

```
## DO this
${Text(_("Click over to {link_start}the home page{link_end}.")).format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
)}
```

python-concat-html

It is safer to use the `HTML()` and `Text()` functions, rather than concatenating strings with HTML. An even better solution would be to handle interpolation with HTML in a proper template, like a Mako template.

Take the following violation as an example.

```
# Do NOT do this
msg = '<html>' + msg + '</html>'
```

Instead, it is possible to properly HTML-escape `msg` as follows.

```
# DO this
msg = HTML('<html>{msg}</html>').format(msg=msg)
```

python-custom-escape

Instead of writing a custom escaping method that replaces < with `<`, use a standard escaping function like `markupsafe.escape()`.

python-deprecated-display-name

The XBlock function `display_name_with_default_escaped` has been deprecated and should not be used. Instead, you must use the call `display_name_with_default` and follow the best practices for proper escaping based on the context.

It might be that `display_name_with_default_escaped` was called from Python while setting up the context for your Mako template. You still must fix this to be `display_name_with_default` and make sure it is properly escaped in the Mako template.

Take note of any places where this value was used in a JavaScript context. You must make sure that this data is properly escaped downstream when it is finally added to the page, for example, in an Underscore.js template.

python-interpolate-html

Interpolation with HTML should use the `HTML()`, `Text()`, and `format()` functions. For details, see *python-concat-html*.

python-parse-error

This violation likely means that there is a syntax error in the Python file. If the Python file is valid, in addition to using the disable pragma detailed in *Disabling Violations*, please report the issue through the Getting Help page on the Open edX portal.

python-requires-html-or-text

In Python, when using either `HTML()` or `Text()` for interpolation with the `format()` function, you must wrap the initial string with `HTML()` or `Text()` as appropriate.

In a Mako expression, any interpolation using `format()` with interpolated `HTML()` calls must be preceded by a call to `Text()`. An expression with interpolation typically does not begin with `HTML()` because in a template, any HTML will be either interpolated in or moved out of the expression and into the outer template HTML.

The following is an example Mako violation.

```
## Do NOT do this
${_("Click over to {link_start}the home page{link_end}.").format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
)}
```

Instead, use `Text()`, as in the following example.

```
## DO this
${Text(_("Click over to {link_start}the home page{link_end}.")).format(
    link_start=HTML('<a href="/home">'),
    link_end=HTML('</a>'),
)}
```

For a deeper understanding of why the function `Text()` is required in the above example, see *Why Do I Need Text() with HTML()?*.

python-wrap-html

When interpolating a string containing HTML using a call to `format()`, you must wrap the HTML with `HTML()`. You might see this issue in a Mako template or a Python file. Also, you might have HTML embedded into a larger string that first needs to be interpolated in. Or, you might already be interpolating in smaller strings containing HTML, but they simply are not yet protected by a call to `HTML()`.

For proper use of `HTML()` and `Text()`, see *HTML Context in Mako*.

underscore-not-escaped

Underscore.js template expressions should all be HTML-escaped using <%- expression %>. The only exceptions where you can use <%= which does not escape is when making an HtmlUtils call.

For more details, see *Underscore.js Template Files*.

10.1.6 Advanced Topics

The following advanced topics cover rare cases and provide a more in-depth explanation of some methods you can use to prevent cross site scripting vulnerabilities.

Why Use Both js_escaped_string and dump_js_escaped_json?

To escape strings in Mako templates, why must we use dump_js_escaped_json in addition to using js_escaped_string?

- The js_escaped_string function provides the additional benefit of returning an empty string in the case of None.
- The js_escaped_string and wrapping quotes makes the expected type more declarative.

Mako Filter Ordering and the n Filter

Mako executes any default filter before it executes filters that are added inside an expression. One such default filter is the decode.utf8 filter, which is used to decode to UTF-8, but only if the Python object is not already unicode.

Take the following example Mako expression.

```
${data | h}
```

When Mako compiles this expression to Python, it is translated to the following Python code.

```
__M_writer(filters.html_escape(filters.decode.utf8(data)))
```

From the Python line above, you can see that the default decode.utf8 filter is applied before the h filter, which was supplied inside the expression.

The n filter can be used to turn off all default filters, including the decode.utf8 filter. Here is an example Mako expression.

```
${data | n}
```

In this case, when Mako compiles this expression to Python, the following Python code is the result.

```
__M_writer(data)
```

For more information, see Mako: Expression Filtering.

Mako Blocks

A Mako `%block` can sometimes create tricky situations where the context is not clear. In these cases, it would be best to provide the context (for example, HTML or JavaScript) in the name of the block.

Take the following Mako `%block` definition as an example.

```
<%page expression_filter="h"/>
...
<%block name="html_title">${display_name}</%block>
```

Based on the above `%block` definition, only the name of the block tells us that it is HTML-escaped, and it is only usable in an HTML context. You could not use this same `%block` in a JavaScript context.

Here is this same `%block` above, as it is actually used to display the title.

```
<title>
    <%block name="html_title"></%block>
</title>
```

For more information, see Mako: Defs and Blocks.

Strings Containing JSON in Mako

In the same way that we wait as long as possible to escape, once we know the context, we also recommend waiting as long as possible before converting from Python to JSON. Mako templates are often the place where the Python object should finally be dumped to JSON.

If you find yourself with a string that already contains JSON inside a Mako template, and you need to use it in a JavaScript context, you have the following two options.

- Where appropriate, you could attempt to refactor the code to move the call to `json.dumps` from the Python file feeding the Mako template, into the Mako template, replacing that call with `dump_js_escaped_json`.
- You can call `json.loads` before dumping it to ensure it is parseable, as in the following example.

```
<script>
    var options = ${json.loads(options_json_string) | n, dump_js_escaped_json};
</script>
```

Why Do I Need Text() with HTML()?

You might wonder why the `Text()` function is required in Mako templates to make the `HTML()` function work.

The magic behind the `Text()` and `HTML()` functions is a library called `markupsafe` and its `Markup` class, which designates that a string is HTML markup and no longer needs to be HTML-escaped. The difference between `Text()` and `HTML()` is that `Text()` HTML-escapes before it becomes `Markup`, where `HTML()` simply marks a string as `Markup`.

The magic of `Markup` is that any string that is formatted into it is HTML- escaped during that process. Note how the & is HTML-escaped in the following example.

```
>>> from markupsafe import Markup
>>> Markup('<div>{}</div>').format('Rock & Roll')
Markup(u'<div>Rock &amp;amp; Roll</div>')
```

For the next example, when `Markup` is formatted into a `Markup` object, it understands that it should not be HTML-escaped and thus the & will remain unchanged.

```
>>> Markup('<div>{}</div>').format(Markup('Rock & Roll'))
Markup(u'<div>Rock & Roll</div>')
```

A problem arises when we use format on a plain string. Since a string does not know anything about `Markup`, the result of this is a plain string again, rather than a `Markup` object. Thus, the result has lost its `Markup` magic.

```
>>> '<div>{}</div>'.format(Markup('Rock & Roll'))
'<div>Rock & Roll</div>'
```

In Mako, we add page-level HTML-escaping by default, which also uses the `markupsafe` library. Mako expressions will therefore respect `Markup` objects and will not double escape.

```
<%page expression_filter="h"/>
...
${data}
```

Therefore, the problem with using `HTML()` without the initial `Text()` is that the `Markup` object becomes a plain old string and it ends up getting HTML-escaped, when your intention was to keep the HTML from being HTML-escaped.

10.1.7 Additional Resources

To learn more about XSS in general, see the following references.

- OWASP: Cross-site Scripting (XSS)
- OWASP: XSS (Cross Site Scripting) Prevention Cheat Sheet
- OWASP: DOM based XSS Prevention Cheat Sheet
- OWASP: XSS Filter Evasion Cheat Sheet
- A good article on Input Validation in Python. You must still use proper escaping on output!

10.2 Preventing XSS by Stripping HTML Tags

10.2.1 Overview

Note: Having server-side HTML data was more likely with our legacy code, but is still possible. We should avoid data with HTML where possible.

There are certain cases where you have server-side HTML and don't want to see escaped HTML, but instead want to either:

- Strip all HTML tags from your server-side data, or
- Strip all HTML tags except safe HTML tags (e.g. "<br />") from your server-side data.

In both cases, we use a library called bleach.

10.2.2 Mako filters for bleaching

At the time of writing this, we do not yet have Mako filters for bleaching. However, that would be very useful._ If you need this, please do the following:

1. See if this was already added,
2. If not, implement it and add to the xss-lint repo.
3. Update this documentation to detail using the filters.

10.2.3 Strip all HTML tags

You would typically do this when people have entered HTML tags inside a field in the past, and you no longer want to support HTML, but you also don't want escaped HTML tags to start appearing on the page.

Here is an example using bleach to strip all tags.

10.2.4 Strip all but safe HTML tags

You would do this if you in fact want to allow a user to be able to use certain simple HTML tags, like "<br />", in their input. Use this sparingly. It is much simpler to deal with plain text fields.

Here is an example using bleach to only allow basic/safe supported tags.

In addition to adding in the HTML to your data, you will probably need to turn off HTML escaping when outputting this data inside a template. The following is an example of this in Mako.

```
# Sample Mako template with page level HTML-escaping on by default

# Expression before:

${title}

# Expression after:

# Title was cleaned with bleach and is safe.
${title | n, decode.utf8}
```

10.3 Preventing XSS in Django Templates

Django Templates are safe-by-default, which means that expressions are HTML-escaped by default. However, there are cases where expressions are not properly escaped by default:

1. If your template includes JavaScript, then any expression inside the JavaScript should be JavaScript-escaped and not HTML-escaped. See *JavaScript Context in Mako* for some help on this topic, although it was written for Mako Templates.
2. We would like to HTML-escape translations, but Django Templates assumes translations are safe and thus they are not HTML-escaped. See below for details on how to properly escape translations in Django Templates.

XSS is a security concern. To learn more about XSS in general, what it is, and why we want to prevent it, see Best Practices for Preventing XSS.

Note: Do not use the "striptags" filter, which only makes an attempt at stripping HTML. Instead, use the bleach library.

10.3.1 HTML-escaping Translations in Django Templates

In Django templates, strings wrapped in `trans` and `blocktrans` are not automatically escaped, which leads to a vulnerability where translators could include malicious script tags in their translations.

HTML-escaping Simple Translations

For most cases simply wrapping the `trans` or `blocktrans` in a `force_escape` filter is sufficient.

Trans Example:

Approach 1:

```
{% trans "somestring" as tmsg %}{{ tmsg | force_escape }}
```

Blocktrans Example:

```
{% filter force_escape %}
    {# Translators: Some note here. #}
    {% blocktrans trimmed with organization_name=program_details.organizations.0.display_
↪name platform_name=site.siteconfiguration.platform_name %}
    a program offered by {{ organization_name }}, in collaboration with {{ platform_name␣
↪}}
    {% endblocktrans %}
{% endfilter %}
```

Note: translator notes must be in the line immediately preceding the translated string, so the force_escape filter should be declared around the translator's note as well.

There also exists a **deprecated** custom filter named htmlescape in credentials that does the same thing as force_escape, only with hacky exceptions for some HTML. Instead, please follow the examples documented in this page.

HTML Interpolation and HTML-escaping

In some cases developers mix HTML into strings directly, typically for styling purposes. In these cases simply force escaping the entire translated string is not an option, since it will escape the HTML that is 'safe' and developer written.

To solve this in credentials and credentials-themes, a custom tag was created which allows for interpolating safe-HTML into a string. Below is the custom tag and an example of its usage.

Custom Interpolation Tag Example:

```
{% blocktrans trimmed asvar msg %}
some text {start_anchor} anchor text for to display {end_anchor}.
{% endblocktrans %}
{% interpolate_html msg start_anchor='<ahref="'|add:site.siteconfiguration.certificate_
↪help_url|add:'">'|safe end_anchor='</a>'|safe %}
```

See an example of this custom tag used in the credentials codebase.

The interpolate_html tag is now available in a shared library for use in all IDAs.

10.4 Preventing XSS in React

React is safe-by-default, so there are fewer places where you need to be careful regarding XSS. In general, JSX knows what is HTML and what is not, and properly HTML-escapes whatever is not meant to be markup.

The two places where you need to be more careful, include:

1. Using the aptly named dangerouslySetInnerHTML function.
2. Handling i18n and translations.

10.4.1 i18n and Translations

If you use the library react-intl, it provides several components for handling messages; some are safe and some are not.

Use FormattedMessage

Use FormattedMessage for safe translations. The messages and translated messages will all be properly HTML-escaped. It will also properly handle HTML-escaping with interpolated variables. The code gets a little messy with interpolated variables, and if the interpolations have their own translated message, it may be difficult on translators, but it is workable.

```
<FormattedMessage
  id="test.hello"
  defaultMessage="Hello {name}, please go visit {link}"
  values={{
    name: <strong>Ben</strong>,
    link:(
      <Hyperlink destination="/visit-me"
        content={
          <FormattedMessage
```

(continues on next page)

(continued from previous page)

```
            id="test.hello.link"
            defaultMessage="this link"
          />
        }
      />
    ),
  }}
/>
```

Don't Use FormattedHTMLMessage

FormattedHTMLMessage uses dangerouslySetInnerHTML behind the scenes. It is meant to work with legacy strings that are safe and contain HTML, but it is never safe. Even if the default message is safe, we can't ensure that a translator doesn't input some unsafe text in the translation.

```
// do NOT use this!
<FormattedHTMLMessage
  id="test.evil"
  defaultMessage={ `<img src='x' onError="alert('Bad bad!')" />` }
/>
```

Alternative Message Components

If you run into a use case that cannot be resolved via FormattedMessage, you can explore some of the following:

- https://www.npmjs.com/package/react-intl-formatted-xml-message
- Related Threads:
  - https://github.com/yahoo/react-intl/issues/68#issuecomment-276702602
  - https://github.com/yahoo/react-intl/issues/513
  - Note: These were mentioned in the react-intl-formatted-xml-message README.
- Markdown (react-remarkable)
  - See https://github.com/yahoo/react-intl/issues/513#issuecomment-252083860

CHAPTER

ELEVEN

LANGUAGE STYLE GUIDELINES

11.1 EdX JavaScript Style Guide

This section describes the requirements and conventions used to contribute JavaScript programming to the edX platform.

11.1.1 JavaScript Version

edX JavaScript should be written consistent with the latest ECMA-262 specification in order to ensure future support, the largest community and the availability of modern features. To support this syntax in older browsers, use Babel. Babel may also be configured to add syntax extensions widely adopted by the community of our recommended framework (e.g., JSX).

Note: Much of edX's existing front end code is written conformant to the version of ECMA-262 released in 2009 (ES5). Files written in ES5 should be gradually converted to the newer standard as new development in those feature areas requires.

11.1.2 Code Style

In order to standardize and enforce Open edX's JavaScript coding style across multiple codebases, edX has published an ESLint configuration that provides an enforceable specification. EdX JavaScript style generally follows the Airbnb JavaScript Style Guide, with a few custom rules. The edX ESLint Config is made available as an npm package that can be installed into any Open edX package.

Note: The edX ESLint Config for ES5 may be used where ES5 is in use. Both configs may be used in the same codebase through the use of ESLint glob configurations.

11.1.3 Testing

All JavaScript code should be tested using Jasmine. In addition, there are a number of Jasmine-based helper classes provided by the edX UI Toolkit.

JavaScript tests are run with Karma, along with karma-coverage to provide code coverage reporting.

For more information about testing JavaScript, see the description of testing for the edx-platform repository.

11.1.4 Documentation

All JavaScript code should be documented using JSDoc. For detailed information about using JSDoc, see the JSDOC site.

As a tool, JSDoc takes JavaScript code with special `/** */` comments and produces HTML documentation for it. An example follows.

```
/** @namespace */
var util = {
    /**
     * Repeat <tt>str</tt> several times.
     *
     * @param {string} str The string to repeat.
     * @param {number} [times=1] How many times to repeat the string.
     * @returns {string}
     */
    repeat: function(str, times) {
        if (times === undefined || times < 1) {
            times = 1;
        }
        return new Array(times+1).join(str);
    }
};
```

11.2 EdX Objective-C Style Guide

This section describes the requirements and conventions for contributing Objective-C programming to the edX platform.

11.2.1 Principles

- Favor clarity and simplicity. Remember the principle of least surprise.
- Build strong interface boundaries. Sometimes this results in a little more code. This is okay. For example, do not expose a whole child object just to expose one of its properties. Instead add a trampoline method.
- Work with the compiler and type system not around it. If you are trying to solve a problem and there is a way to get the compiler to check it, do that. Once again, the shortest way to do something is not always the best way to do something.
- Break functionality into unit testable pieces.
- Avoid inheritance. Inheriting from some system classes, like UIViewController and UIView, is of course, necessary and sometimes a class hierarchy is the right pattern, but we strongly prefer to use composition and delegation over inheritance. If you are designing something with inheritance always think about if there's a way to change your design to not use it.

11.2.2 Syntax and Organization

- Follow the rules in Apple's guidelines except where they directly contradict what follows.
- You can automatically apply many of these formatting suggestions by typing `gradle format` into a terminal from the project root directory.
- Method names in Cocoa have a grammar. Compare the `-getObject:atIndex:` method of `NSArray` to a hypothetical `-get:index` method. `-getObject:atIndex:` is much more descriptive and also sounds more natural in English. However, note that this is not the same grammar as English. For example, as described in Apple's documentation, the word `and` preceding an argument is not necessary even though it would be necessary in some cases for the method signature to form an English sentence.
- When in doubt, think about how something would look if it were part of one of Apple's core frameworks such as UIKit.
- `#import` declarations should be in the following groups, ordered alphabetically within each group:

    1. System headers (including third party libraries)
    2. Header for current file (only if in a .m file)
    3. Project headers

    For example,

```
// MyClass.m

#import <FacebookSDK/FacebookSDK.h>
#import <QuartzCore/QuartzCore.h>

#import "MyClass.h"

#import "OEXSomething.h"
#import "OEXWhatever.h"
```

- Avoid `#import` in headers where possible. This makes dependencies more explicit and results in fewer headers included. This improves compilation speeds. Instead use the `@class` and `@protocol` forward declarations.

```
// Good
@class SomeClass;

@interface OtherClass
- (instancetype)initWithSomeClass:(SomeClass*)object;
@end


// Bad
#import "SomeClass.h"

@interface OtherClass
- (instancetype)initWithSomeClass:(SomeClass*)object;
@end
```

- Avoid `#define`. Instead use constant declarations. They are more accessible to the compiler. For example, "Quick Open" does not work with macros. Additionally, macro functions are error prone and hard to debug. Some examples follow:

```
extern NSString* const OEXExampleKey = @"OEXExampleKey";
extern CGFloat const OEXExampleConstant = 3;
```

  For C macros with logic, consider if you can just use a regular function instead of the macro.

- Constants that are local to a single file should be declared `static`.
- Use spaces for indentation instead of tabs. This is the Xcode default.
- Prefer properties over bare instance variables. In general, you should only mention an `ivar` in a setter for that property.

```
// Good
@interface SomeClass

@property (strong, nonatomic) NSString* foo;

@end


// Bad
@interface SomeClass {
    NSString* _foo;
}
@end

// Worse
@interface SomeClass {
    NSString* foo;
}
@end
```

- Use a leading underscore to name an `ivar`. However, you should favor properties and auto synthesis and almost never refer to an `ivar` explicitly. Sometimes you do need to synthesize an `ivar` explicitly, for example when implementing a protocol. Again, those should use leading underscores.

```
@synthesize something = _something;
```

- Do not bother with `@synthesize` for autosynthesized properties.
- Private methods do not need a leading prefix like _ or `p_`. Their private nature is implied by their absence from a class's header file.
- Methods added in categories to system libraries should be prefixed `oex_` (for Open edX). Categories have a flat namespace. Using a prefix means our additions will not interfere with any other libraries.
- **Follow the standard Cocoa file naming conventions:**
  - Class `Example` should be in `OEXExample.[hm]`
  - Category `SomethingAdditions` on class `OEXExample` should be in `OEXExample+SomethingAdditions.[hm]`
  - A view controller for the `Example` screen should be in `OEXExampleViewController.[hm]`
  - A view that displays an `Example` should be in `OEXExampleView.[hm]`
- **Categories should be named for the functionality they provide.**

```
// Good
@interface NSString (OEXFormattingAdditions)
//... functions that control formatting
@end


// Bad
@interface NSString (OEXHelpers)
// ... functions that do many different kinds of things
@end
```

- Delegate methods should include a sender as the first argument. This allows the owner to distinguish which object is sending the message and sometimes to avoid having an extra `ivar`.

```
// Good
@interface SomeClassDelegate
- (void)tabView:(TabView*)tabView choseTabAtIndex:(NSUInteger)index;
@end


// Bad
@interface SomeClassDelegate
- (void)choseTabAtIndex:(NSUInteger)index;
@end
```

- Only put properties and methods in headers that need to be part of a class's interface. Everything else should be declared in a class continuation in the implementation file.
- Avoid lazy initialization of properties. Otherwise, it is hard to reason about property accesses. With lazy initialization, even read only objects have complicated threading behavior.

```
// Bad
@interface SomeClass
@property (strong, nonatomic) OtherClass* field;
@end
```

(continues on next page)

(continued from previous page)

```
@implementation SomeClass

- (OtherClass*)field {
    if(_field == nil) {
        _field = [[OtherClass alloc] init];
    }
    return _field;
}

@end
```

Instead, add an explicit creation function like `makeFieldIfNecessary` or just instantiate it in `-init`. For expensive things, the caller should have control, and for cheap things you are not gaining any performance advantage for the cost of decreased determinism.

- Avoid Key Value Observing. It is occasionally the only way to observe something, but do not design interfaces that use it. It is an error prone API.
- Do not use exceptions for control flow. They should only be for top level failure conditions indicating programmer error. ARC is not thread safe by default and Swift does not even have exceptions.
- Use line comments (`//`) instead of block comments (`/* */`). They are easier to stack and Xcode has a keyboard shortcut for them (`-/`).
- Use triple slash comments (`///`) to create inline documentation. For example:

```
/// Method that does a thing
- (void)someMethod { }
```

- Always comment the type of the contents of collection types like `NSArray` and `NSDictionary`. This makes the expectations of the code clear. For example:

```
@interface SomeClass

/// Contents are NSString*
@property (copy, nonatomic) NSArray* elements;
@end
```

- Comparisons should be explicit for when checking pointers for null. For example:

```
// Good
SomeObject* object = ...;
if(object == null) {


// Bad
SomeObject* object = ...;
if(!object) {
```

- Separate binary operands with a single space, but unary operands and casts with none.

```
1 + 2   // Good
1+1     // Bad
1+ 1    // Bad
```

(continues on next page)

(continued from previous page)

```
-3      // Good
- 3     // Bad
```

- Always use braces on control structures, even if they are optional. For example:

```
// Good
if(someCondition) {
    aSingleLine();
}

// Bad
if(someCondition) aSingleLine();
```

- Properties should be marked `nonatomic` unless there is a very good reason otherwise. Marking a property `atomic` should signal that you have thought hard about the threading behavior of this property and very intentionally decided that it should work through `atomic` properties and not by isolating access to a queue.
- Declare memory semantics. All properties should be marked `strong`, `weak`, or `assign`. There are defaults for different types that are usually right, but making it explicit forces you to think about whether you are creating cycles in memory.

```
// Good
@property (strong, nonatomic) SomeObject* foo;

// Bad
@property SomeObject* foo;
```

11.2.3 Writing Tests

- Unit test files are typically oriented around testing a single file. The name of a test file should be the name of the file being tested but with the word `Tests` at the end. As an example, a test file for `OEXSomeClass.m` is `OEXSomeClassTests.m`
- Tests should always run against test data, not a current user's. This means that after the tests are over, it should be as if they never ran.
- Network data should always be mocked. The tests should have the exact same result whether or not an Internet connection is available to the test runner.
- If you need to expose a method just for testing, prefix it `t_`. This indicates that it should only be used by test code. This will often come up with view tests since their programmatic interface is often much simpler than their UI contract. When exposing such methods, you should ensure that a refactor or redesign of that view should not invalidate the test.

  For example, a login screen might have a `t_tapLogin` method that triggers the action of the login button. Even if the login screen is refactored or redesigned it will probably still have a login button that can be tapped so it is safe to make this part of the contract. However, this logic does not extend to the login button itself. There are a number of ways to implement what appears to the user as a button, such as gesture recognizers and overriding `touchesBegan:`, so exposing a `t_loginButton` method returning a `UIButton` would violate this rule.
- Do not redeclare a method as public inside the test. This is fragile since changes will not be caught by the compiler.

```
// Good
// SomeClass.h
@interface SomeClass
@end

@interface SomeClass (Testing)
- (BOOL)t_isVisible;
@end

// SomeClass.m
@implementation SomeClass (Testing)
- (BOOL)t_isVisible {
    return [self isVisible];
}
@end


// Bad
// SomeClass.h
@interface SomeClass
@end

// SomeClass.m
@implementation SomeClass
- (void)isVisible {
    ...
}
@end

// SomeClassTests.m
@interface SomeClass (Testing)
- (void)isVisible;
@end
```

11.3 EdX Python Style Guide

This section describes the requirements and conventions used to contribute Python programming to the edX platform.

- *References*

11.3.1 Principles

Generally, do not edit files just to change the style. But do aim for this style with new or modified code.

See also *Code Quality*.

Write a Good repr() for Each Class

Each class should have a __repr__() method defined, so that calling *repr()* on an instance of the class returns something meaningful that distinguishes objects from each other to a human being. This is useful for debugging purposes.

11.3.2 Syntax and Organization

Follow PEP 8.

- 4-space indents (no tabs)
- Names like this: modules_and_packages, functions_and_methods, local_variables, GLOBALS, CONSTANTS, MultiWordClasses
- Acronyms should count as one word: RobustHtmlParser, not RobustHTMLParser
- Trailing commas are good: they prevent having to edit the last line in a list when adding a new last line. You can use them in lists, dicts, function calls, etc.
- EXCEPT: we aren't (yet) limiting code lines to 79 characters. Use 120 as a limit for code. Please use 79 chars as a limit for docstring lines though, so that the text remains readable.

- *Breaking Long Lines*
- *Imports Order*

Breaking Long Lines

Follow these guidelines:

- Try to refactor the code to not need such long lines. This is often the best option, and is often overlooked for some reason. More, shorter lines are good.
- If you need to break a function call over more than one line, put a newline after the open paren, and move the arguments to their own line. DO NOT indent everything to where the open paren is. This makes the code too indented, and makes different function calls near each other indented different amounts.

NO NO NO!:

```
results = my_object.some_method(arg1,    # this is very
                                arg2,    # very ugly and makes
                                arg3,    # the code squished over on the right.
                                )
```

YES:

```
results = my_object.some_method(
    arg1,
    arg2,
    arg3,
)
```

OR:

```
results = my_object.some_method(
    arg1, arg2, arg3
)
```

Important points:

- Do not over-indent to make things line up with punctuation on the first line.
- Closing paren should be on a line by itself, indented the same as the first line.
- The first line ends with the open paren.

Imports Order

PEP8 recommends a most-general to most-specific import order, which means this order:

- Standard library imports
- Third Party Library imports
- Local imports

Alphabetize each group of imports, and use a single blank line to separate groups.

Note: Most Open edX repositories use the isort library, which will automatically order imports to follow PEP8.

11.3.3 Pylint Guidelines and Practices

- For unused args, you can prefix the arguments with an underscore (_) to mark them as unused (as convention), and pylint will accept that.
- Adding a TODO in one place requires you to make a pylint fix in another (just to force us to clean up more code).
- No bare `except` clauses. `except:` should be `except Exception:`, which will prevent it from catching system-exiting exceptions, which we probably should not be doing anyway. If we need to, we can catch `BaseException` (There's no point in catching `BaseException`, that includes the exceptions we didn't want to catch with `except:` in the first place.) (ref: http://docs.python.org/2/library/exceptions.html#bltin-exceptions). Catching `Exception`, however, will still generate a Pylint warning ("W0703: catching too general exception.") If you still feel that catching `Exception` is justified, silence the pylint warning with a pragma: `# pylint: disable=broad- except`.
- Although we try to be vigilant and resolve all quality violations, some Pylint violations are just too challenging to resolve, so we opt to ignore them via use of a pragma. A pragma tells Pylint to ignore the violation in the given line. An example is:

```
self.assertEquals(msg, form._errors['course_id'][0])  # pylint: disable=protected-
↪access
```

The pragma starts with a # two spaces after the end of the line. We prefer that you use the full name of the error (`pylint: disable=unused-argument` as opposed to `pylint: disable=W0613`), so that it is more clear what you are disabling in the line.

Classes Versus Dictionaries

It's better to use a class or a `namedtuple` to pass around data that has a fixed shape than to use a `dict`. It makes it easier to debug (because there is a fixed, named set of attributes), and it helps prevent accidental errors of either setting new attributes into the dictionary (which might, for instance, get serialized unexpectedly), or might be typos.

11.3.4 Docstrings

Follow PEP 257.

- Write docstrings for all modules, classes, and functions.
- Always format docstrings using the multi-line convention, even if there's only one line of content (see below).
- Use three double-quotes for all docstrings.
- Start with a one-line summary. If you can't fit a summary onto one line, think harder, or refactor the code.
- Write in Sphinx-friendly prose style. Put double backquotes around code names (variables, parameters, methods, etc).

The preferred style is so-called "Google Style" with readable headers for different sections, and all arguments and return values defined.

Note: There is one exception to the preferred style. REST APIs created using Django REST Framework (DRF) must use a hybrid format that is suitable both for DRF and ReadTheDocs. For more information see the edX REST API Conventions.

For additional information see these references.

- Google Python Style Guide
- Example Google Style Python Docstrings (from Sphinx)

Here's how you write documentation in a mostly "Google Style" manner:

```
def func(arg1, arg2):
    """
    Summary line.

    Extended description of function.

    Arguments:
        arg1 (int): Description of arg1
        arg2 (str): Description of arg2

    Returns:
        bool: Description of return value
    """
```

Note: There are some exceptions:

- The summary line is on the second line, including single-line comments (see below)
- Use the full word "Arguments", although "Args" is also acceptable.

Most of our code is written using an older style:

```
def calculate_grade(course, student):
    """
    Sum up the grade for a student in a particular course.

    Navigates the entire course, adding up the student's grades. Note that
    blah blah blah, and also beware that blah blah blah.

    `course` is an `EdxCourseThingy`. The student must be registered in the
    course, or a `NotRegistered` exception will be raised.

    `student` is an `EdxStudentThingy`.

    Returns a dict with two keys: `total` is a float, the student's total
    score, and `outof` is the maximum possible score.
    """
```

If you only have a single line in your docstring, first consider that this is almost certainly not enough documentation, and write some more. But if you do have just one line, format it in a similar way to a multi-line docstring:

```
def foo(a, b):
    """
    Computes the foo of a and b.
    """
```

Not like this:

```
def foo(a, b):
    """Computes the foo of a and b.""" # NO NO NO
```

We intentionally stray from PEP 257 in this case. The formatting inconsistency between single and multi-line docstrings can result in merge conflicts when upstream and downstream branches change the same docstring. See this GitHub comment for more context.

11.3.5 References

- PEP 8
- PEP 257
- Google Python Style Guide
- Django Coding Style
- The Hitchhiker's Guide to Python
- Pythonic Sensibilities

11.4 EdX Sass Style Guide

This section describes the requirements and conventions used to contribute Sass stylesheets to the edX platform.

11.4.1 Code Style

In order to standardize and enforce Open edX's Sass coding style across multiple codebases, edX uses Stylelint which is a widely adopted CSS linter written in JavaScript. In particular, edX provides the edX Stylelint Config which is an npm package that defines the rule set to be used to validate Sass.

EdX generally adopts the standard Stylelint rule set:

- CSS Rules
- SCSS-Specific Rules

If you are interested in the exceptions, see the edX Stylelint Config README.

11.4.2 Use Variables

It is strongly recommended to avoid hard-coding values such as colors and fonts. Using hard-coded values makes it difficult to change the value in the future as it may have been used in many stylesheets. Using variables also allows themes to simply override the value in one place.

Before defining a new variable, see if you can reuse an existing one. If you need to create a new one, be sure to use the `!default` flag. This allows themes to provide a different value for this variable if they choose. See the Sass documentation for default flag for more details.

For example, here is an example of a hard-coded style:

```
.my-element {
  color: #0000ff;
}
```

This should instead be written as:

```
$my-custom-color: #0000ff !default;

.my-element {
  color: $my-custom-color;
}
```

CHAPTER

TWELVE

GLOSSARY

A - C - D - E - F - G - H - I - K - L - M - N - O - P - R - S - T - V - W - XYZ

Note: Most of the links to documentation provided in this glossary are to the Building and Running an edX Course guide, for edX partners. Many of the same topics are available in the Open edX version of this guide, Building and Running an Open edX Course.

12.1 A

AAC

Advanced audio coding (AAC) is an audio coding standard for digital audio compression. AAC is the standard format for YouTube.

A/B test

See Content Experiment.

About page

The course page that provides potential learners with a course summary, prerequisites, a course video and image, and important dates.

accessible label

In a problem component, you use special formatting to identify the specific question that learners will answer by selecting options or entering text or numeric responses.

This text is referred to as the accessible label because screen readers read all of the text that you supply for the problem and then repeat the text that is identified with this formatting immediately before reading the answer choices for the problem. This text is also used by reports and Insights to identify each problem.

All problems require accessible labels.

For more information, see The Simple Editor.

advanced editor

An OLX (open learning XML) editor in a problem component that allows you to create and edit any type of problem. For more information, see The Advanced Editor.

Amazon Web Services (AWS)

A third-party file hosting site where course teams can store course assets, such as problem files and videos. If videos are posted on both YouTube and AWS, the AWS version of the video serves as a backup in case the YouTube video does not play.

assignment type

The category of graded student work, such as homework, exams, and exercises. For more information, see Establishing a Grading Policy For Your Course.

12.2 C

CAPA problem

A CAPA (computer assisted personalized approach) problem refers to any of the problem types that are implemented in the edX platform by the `capa_module` XBlock. Examples range from text input, drag and drop, and math expression input problem types to circuit schematic builder, custom JavaScript, and chemical equation problem types.

Other assessment methods are also available, and implemented using other XBlocks. An open response assessment is an example of a non-CAPA problem type.

certificate

A document issued to an enrolled learner who successfully completes a course with the required passing grade. Not all edX courses offer certificates, and not all learners enroll as certificate candidates.

For information about setting up certificates for your course, see Setting Up Certificates in Studio.

chapter

See *Section*.

checkbox problem

A problem that prompts learners to select one or more options from a list of possible answers. For more information, see Checkbox Problem.

chemical equation response problem

A problem that allows learners to enter chemical equations as answers. For more information, see Chemical Equation Problem.

circuit schematic builder problem

A problem that allows learners to construct a schematic answer (such as an electronics circuit) on an interactive grid. For more information, see Circuit Schematic Builder Problem.

closed captions

The spoken part of the transcript for a video file, which is overlaid on the video as it plays. To show or hide closed captions, you select the **CC** icon. You can move closed captions to different areas on the video screen by dragging and dropping them.

For more information, see learners:Video Player.

codec

A portmanteau of "code" and "decode". A computer program that can encode or decode a data stream.

cohort

A group of learners who participate in a class together. Learners who are in the same cohort can communicate and share experiences in private discussions.

Cohorts are an optional feature of courses on the edX platform. For information about how you enable the cohort feature, set up cohorts, and assign learners to them, see Using Cohorts in Your Courses.

component

The part of a unit that contains your actual course content. A unit can contain one or more components. For more information, see Developing Course Components.

content experiment

You can define alternative course content to be delivered to different, randomly assigned groups of learners. Also known as A/B or split testing, you use content experiments to compare the performance of learners who have been exposed to different versions of the content. For more information, see Overview of Content Experiments.

content library

See *Library*.

content-specific discussion topic

A category within the course discussion that appears at a defined point in the course to encourage questions and conversations. To add a content- specific discussion topic to your course, you add a discussion component to a unit. Learners cannot contribute to a content-specific discussion topic until the release date of the section that contains it. Content-specific discussion topics can be divided by cohort, so that learners only see and respond to posts and responses by other members of the cohort that they are in.

For more information, see Working with Discussion Components. For information about making content-specific discussion topics divided by cohort, see Setting up Discussions in Courses with Cohorts.

course catalog

The page that lists all courses offered in the edX learning management system.

course handouts

Course handouts are files you make available to learners on the **Home** page. For more information, see Adding Course Updates and Handouts.

course mode

See *enrollment track*.

course navigation pane

The navigation frame that appears at one side of the **Course** page in the LMS. The course navigation pane shows the sections in the course. When you select a section, the section expands to show subsections. When you select a subsection, the first unit in that subsection appears on the course page.

See also *Unit Navigation Bar*.

Course page

The page that opens first when learners access your course. On the **Course** page, learners can view the course outline and directly access the course, either by clicking a specific section or subsection on the outline, or by clicking the **Start Course** button (**Resume Course** if the learner has previously accessed the course).

The latest course update, such as a course welcome message, appears above the course outline. Links to various **Course Tools** including **Bookmarks**, **Reviews** and **Updates** appear at the side of this page. This page is a combination of the former **Home** and **Courseware** pages.

course run

A version of the course that runs at a particular time. Information about a course run includes start and end dates, as well as staff and the languages the course is available in. You can create a course run when you create a course.

course track

See *enrollment track*.

courseware

In OLX (open learning XML) and in data packages, "courseware" refers to the main content of your course, consisting mainly of lessons and assessments. Courseware is organized into sections, subsections, units, and components. Courseware does not include handouts, the syllabus, or other course materials.

Note that the **Course** page was formerly called the **Courseware** page.

course-wide discussion topic

Optional discussion categories that you create to guide how learners find and share information in the course discussion. Course-wide discussion topics are accessed from the **Discussion** page in your course. Examples of course-wide discussion topics include Announcements and Frequently Asked Questions. Learners can contribute to these topics as soon as your course starts. For more information, see Creating Course Discussions and Create Course-Wide Discussion Topics.

If you use cohorts in your course, you can divide course-wide discussion topics by cohort, so that although all learners see the same topics, they only see and respond to posts and responses by other members of the cohort that they are in. For information about configuring discussion topics in courses that use cohorts, see Setting up Discussions in Courses with Cohorts.

custom response problem

A custom response problem evaluates text responses from learners using an embedded Python script. These problems are also called "write-your-own-grader" problems. For more information, see Write-Your-Own-Grader Problem.

12.3 D

data czar

A data czar is the single representative at a partner institution who is responsible for receiving course data from edX, and transferring it securely to researchers and other interested parties after it is received.

For more information, see the Using the edX Data Package.

discussion

The set of topics defined to promote course-wide or unit-specific dialog. Learners use the discussion topics to communicate with each other and the course team in threaded exchanges. For more information, see Creating Course Discussions.

discussion component

Discussion topics that course teams add directly to units. For example, a video component can be followed by a discussion component so that learners can discuss the video content without having to leave the page. When you add a discussion component to a unit, you create a content-specific discussion topic. See also *Content Specific Discussion Topic*.

For more information, see Working with Discussion Components.

discussion thread list

The navigation frame that appears at one side of the **Discussion** page in the LMS. The discussion thread list shows the discussion categories and subcategories in the course. When you select a category, the list shows all of the posts in that category. When you select a subcategory, the list shows all of the posts in that subcategory. Select a post to read it and its responses and comments, if any.

dropdown problem

A problem that asks learners to choose from a collection of answer options, presented as a drop-down list. For more information, see Dropdown Problem.

12.4 E

edX101

An online course about how to create online courses. The intended audience for edX101 is faculty and university administrators.

edX Edge

edX Edge is a less restricted site than edX.org. While only edX employees and consortium members can create and post content on edX.org, any users with course creator permissions for Edge can create courses with Studio on studio.edge.edx.org, then view the courses on the learning management system at edge.edx.org.

edX Studio

The edX tool that you use to build your courses. For more information, see Getting Started with Studio.

embargo

An embargo is an official ban on trade or commercial activity with a particular country. For example, due to U.S. federal regulations, edX cannot offer certain courses (for example, particular advanced STEM courses) on the edx.org website to learners in embargoed countries. Learners cannot access restricted courses from an embargoed country. In some cases, depending on the terms of the embargo, learners cannot access any edX courses at all.

enrollment mode

See *enrollment track*.

enrollment track

Also called **certificate type**, **course mode**, **course seat**, **course track**, **course type**, **enrollment mode**, or **seat type**.

The enrollment track specifies the following items about a course.

- The type of certificate, if any, that learners receive if they pass the course.
- Whether learners must verify their identity to earn a certificate, using a webcam and a photo ID.
- Whether the course requires a fee.
- **audit**: This is the default enrollment track when learners enroll in a course. This track does not offer certificates, does not require identity verification, and does not require a course fee.
- **professional**: This enrollment track is only used for specific professional education courses. The professional enrollment track offers certificates, requires identity verification, and requires a fee. Fees for the professional enrollment track are generally higher than fees for the verified enrollment track. Courses that offer the professional track do not offer a free enrollment track.

  Note: If your course is part of a MicroMasters or professional certificate program, your course uses the verified track. These courses do not use the professional enrollment track.

- **verified**: This enrollment track offers verified certificates to learners who pass the course, verify their identities, and pay a required course fee. A course that offers the verified enrollment track also automatically offers a free non-certificate enrollment track.

- **honor**: This enrollment track was offered in the past and offered an honor code certificate to learners who pass the course. This track does not require identity verification and does not require a fee. Note, however, that as of December 2015, edx.org no longer offers honor code certificates. For more information, see News About edX Certificates.

exercises

Practice or practical problems that are interspersed in edX course content to keep learners engaged. Exercises are also an important measure of teaching effectiveness and learner comprehension. For more information, see Adding Exercises and Tools.

export

A tool in edX Studio that you use to export your course or library for backup purposes, or so that you can edit the course or library directly in OLX format. See also *Import*.

For more information, see Export a Course or Export a Library.

12.5 F

forum

See *Discussion*.

fps

Frames per second. In video, the number of consecutive images that appear every second.

12.6 G

grade range

Thresholds that specify how numerical scores are associated with grades, and the score that learners must obtain to pass a course.

For more information, see Set the Grade Range.

grading rubric

See *Rubric*.

12.7 H

H.264

A standard for high definition digital video.

Home page

See Course Page.

Text component

A type of component that you can use to add and format text for your course. A Text component can contain text, lists, links, and images. For more information, see Working with Text Components.

12.8 I

Image mapped input problem

A problem that presents an image and accepts clicks on the image as an answer. For more information, see Image Mapped Input Problem.

Import

A tool in Studio that you use to load a course or library in OLX format into your existing course or library. When you use the Import tool, Studio replaces all of your existing course or library content with the content from the imported course or library. See also *Export*.

For more information, see Import a Course or Import a Library.

instructor dashboard

A user who has the Admin or Staff role for a course can access the instructor dashboard in the LMS by selecting **Instructor**. Course team members use the tools, reports, and other features that are available on the pages of the instructor dashboard to manage a running course.

For more information, see Managing a Running Course.

12.9 K

keyword

A variable in a bulk email message. When you send the message, a value that is specific to the each recipient is substituted for the keyword.

12.10 L

label

See *Accessible Label*.

LaTeX

A document markup language and document preparation system for the TeX typesetting program. In edX Studio, you can import LaTeX code.

learning management system (LMS)

The platform that learners use to view courses, and that course team members use to manage learner enrollment, assign team member privileges, moderate discussions, and access data while the course is running.

learning sequence

See *Unit Navigation Bar*.

left pane

See *Course Navigation Pane*.

library

A pool of components for use in randomized assignments that can be shared across multiple courses from your organization. Course teams configure randomized content blocks in course outlines to reference a specific library of components, and randomly provide a specified number of problems from that content library to each learner.

For more information, see Working with Content Libraries and Randomized Content Blocks.

live mode

A view that allows the course team to review all published units as learners see them, regardless of the release dates of the section and subsection that contain the units. For more information, see Viewing Published and Released Content.

LON-CAPA

The Learning Online Network with Computer-Assisted Personalized Approach e-learning platform. The structure of CAPA problem types in the edX platform is based on the LON-CAPA assessment system, although they are not compatible.

See also *CAPA Problems*.

12.11 M

math expression input problem

A problem that requires learners to enter a mathematical expression as text, such as e=m*c^2.

For more information, see learners:Math Formatting in the *EdX Learner's Guide*.

MathJax

A LaTeX-like language that you use to write equations. Studio uses MathJax to render text input such as x^2 and sqrt(x^2-4) as "beautiful math."

For more information, see Using MathJax for Mathematics.

module

An item of course content, created in an XBlock, that appears on the **Course** page in the edX learning management system. Examples of modules include videos, HTML-formatted text, and problems.

Module is also used to refer to the structural components that organize course content. Sections, subsections, and units are modules; in fact, the course itself is a top-level module that contains all of the other course content as children.

multiple choice problem

A problem that asks learners to select one answer from a list of options. For more information, see Multiple Choice Problem.

12.12 N

NTSC

National Television System Committee. The NTSC standard is a color encoding system for analog videos that is used mostly in North America.

numerical input problem

A problem that asks learners to enter numbers or specific and relatively simple mathematical expressions. For more information, see Numerical Input Problem.

12.13 O

OLX

OLX (open learning XML) is the XML-based markup language that is used to build courses on the Open edX platform.

For more information, see What is Open Learning XML?.

open response assessment

A type of assignment that allows learners to answer with text, such as a short essay and, optionally, an image or other file. Learners then evaluate each others' work by comparing each response to a *rubric* created by the course team.

These assignments can also include a self assessment, in which learners compare their own responses to the rubric, or a staff assessment, in which members of course staff evaluate learner responses using the same rubric.

For more information, see Introduction to Open Response Assessments.

12.14 P

pages

Pages organize course materials into categories that learners select in the learning management system. Pages provide access to the course content and to tools and uploaded files that supplement the course. Links to each page appear in the course material navigation bar.

For more information, see Managing the Pages in Your Course.

PAL

Phase alternating line. The PAL standard is a color encoding system for analog videos. It is used in locations such as Brazil, Australia, south Asia, most of Africa, and western Europe.

partner manager

Each EdX partner institution has an edX partner manager. The partner manager is the primary contact for the institution's course teams.

pre-roll video

A short video file that plays before the video component selected by the learner. Pre-roll videos play automatically, on an infrequent schedule.

For more information, see Adding a Pre-Roll Video to Your edX Course.

preview mode

A view that allows you to see all the units of your course as learners see them, regardless of the unit status and regardless of whether the release dates have passed.

For more information, see Previewing Draft Content.

problem component

A component that allows you to add interactive, automatically graded exercises to your course content. You can create many different types of problems.

For more information, see Working with Problem Components and Adding Exercises and Tools.

proctored exam

At edX, proctored exams are timed, impartially and electronically monitored exams designed to ensure the identity of the test taker and determine the security and integrity of the test taking environment. Proctored exams are often required in courses that offer verified certificates or academic credit. For more information, see Managing Proctored Exams.

program

A program is a collection of related courses. Learners enroll in a program by enrolling in any course that is part of a program, and earn a program certificate by passing each of the courses in the program with a grade that qualifies them for a verified certificate.

Several types of program are available on edx.org, including MicroMasters, Professional Certificate, and XSeries programs.

program offer

A program offer is a discount offered for a specific program. The discount can be either a percentage amount or an absolute (dollar) amount.

Progress page

The page in the learning management system that shows learners their scores on graded assignments in the course. For more information, see learners:SFD Check Progress in the *EdX Learner's Guide*.

12.15 Q

question

A question is a type of post that you or a learner can add to a course discussion topic to bring attention to an issue that the discussion moderation team or learners can resolve.

For more information, see Creating Course Discussions.

12.16 R

Research Data Exchange (RDX)

An edX program that allows participating partner institutions to request data for completed edx.org courses to further approved educational research projects. Only partner institutions that choose to participate in RDX contribute data to the program, and only researchers at those institutions can request data from the program.

For more information, see Using the Research Data Exchange Data Package.

rubric

A list of the items that a learner's response should cover in an open response assessment. For more information, see the Rubric topic in Introduction to Open Response Assessments.

See also *Open Response Assessment*.

12.17 S

seat type

See *enrollment track*.

section

The topmost category in your course outline. A section can represent a time period or another organizing principle for course content. A section contains one or more subsections.

For more information, see Developing Course Sections.

sequential

See *Subsection*.

short description

The description of your course that appears on the edX Course List page.

For more information, see Course Short Description Guidelines.

simple editor

The graphical user interface in a problem component that contains a toolbar for adding Markdown formatting to the text you supply. The simple editor is available for some problem types. For more information, see Editing a Problem in Studio.

single sign-on (SSO)

SSO is an authentication service that allows a user to access multiple related applications, such as Studio and the LMS, with the same username and password. The term SSO is sometimes used to refer to third party authentication, which is a different type of authentication system. For information about third party authentication, see *Third Party Authentication*.

special exam

A general term that applies to proctored and timed exams in edX courses. See *Timed Exam* and *Proctored Exam*.

split test

See *Content Experiment*.

subsection

A division in the course outline that represents a topic in your course, such as a lesson or another organizing principle. Subsections are defined inside sections and contain units.

For more information, see Developing Course Subsections.

12.18 T

text input problem

A problem that asks learners to enter a line of text, which is then checked against a specified expected answer.

For more information, see Text Input Problem.

timed exam

Timed exams are sets of problems that a learner must complete in the amount of time you specify. When a learner begins a timed exam, a countdown timer displays, showing the amount of time allowed to complete the exam. If needed, you can grant learners additional time to complete the exam. For more information, see Offering Timed Exams.

third party authentication

A system-wide configuration option that allows users who have a username and password for one system, such as a campus or institutional system, to log in to that system and automatically be given access to the LMS. These users do not enter their system credentials in the LMS.

For more information about how system administrators can integrate an instance of Open edX with a campus or institutional authentication system, see Enabling Third Party Authentication.

transcript

A text version of the content of a video. You can make video transcripts available to learners.

For more information, see Obtain a Video Transcript.

12.19 U

unit

A unit is a division in the course outline that represents a lesson. Learners view all of the content in a unit on a single page.

For more information, see Developing Course Units.

unit navigation bar

The horizontal control that appears at the top of the **Course** page in the LMS. The unit navigation bar contains an icon for each unit in the selected subsection. When you move your pointer over one of these icons, the name of the unit appears. If you have bookmarked a unit, the unit navigation bar includes an identifying flag above that unit's icon.

See also *Course Navigation Pane*.

12.20 V

VBR

Variable bit rate. The bit rate is the number of bits per second that are processed or transferred. A variable bit rate allows the bit rate to change according to the complexity of the media segment.

vertical

See *Unit*.

video component

A component that you can use to add recorded videos to your course.

For more information, see Working with Video Components.

12.21 W

whitelist

In edX courses, a whitelist is a list of learners who are being provided with a particular privilege. For example, whitelisted learners can be specified as being eligible to receive a certificate in a course, regardless of whether they would otherwise have qualified based on their grade.

In the grade report for a course, whitelisted learners have a value of "Yes" in the **Certificate Eligible** column, regardless of the grades they attained. For information about the grade report, see Interpreting the Grade Report.

wiki

The page in each edX course that allows both learners and members of the course team to add, modify, or delete content. Learners can use the wiki to share links, notes, and other helpful information with each other. For more information, see Using the Course Wiki.

12.22 XYZ

XBlock

EdX's component architecture for writing course components: XBlocks are the components that deliver course content to learners.

Third parties can create components as web applications that can run within the edX learning management system. For more information, see Open edX XBlock Tutorial.

XSeries

A set of related courses in a specific subject. Learners qualify for an XSeries certificate when they pass all of the courses in the XSeries. For more information, see XSeries Programs.