UNLOCKING LTI

How We Decoupled the LTI 1.3 Launch from the Open edX Platform

Michael Roytman
edX/2U Software Engineer

Welcome to “Unlocking LTI”. Today, we’ll be talking about decoupled LTI 1.3 launches. We’ll cover what that even means, why it’s important, and how you can leverage LTI from your application.

I feel that LTI has garnered quite a bit more attention in the last year or so, at least at 2U, so the objective of this talk is to make you more familiar with the state of our LTI implementation and what the vision is for our use of LTI on the platform.

Now, if you went to David Joy’s talk on “An Opinionated Vision for Open edX Extensibility and Customization”, you may recall that he described the LTI specifications as being “not fit for human consumption.” And, for that reason, I’m not going to spend any time reviewing our LTI implementation code today. Instead, I’ve structured today’s talk using diagrams as an exploration of how LTI was implemented on the platform, the issues we encountered using it, and how we addressed those issues in a way that we believe greatly benefits the platform.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

Here is an overview of the structure of today's talk. You've probably come to today's talk with a series of questions based on the title of the talk alone, and you may see some of them on the screen. I'm planning on answering these questions in the next 35 minutes or so. Of course, I'll leave 10 or so minutes at the end for any remaining questions you may have.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

So, let's start with the question "What is LTI?" I know many of you may be familiar with LTI already, but some of you may not be. It's important that you understand how LTI works generally so that you can better understand and appreciate the problem space, our LTI implementation architecture, and our solution design.

LTI REVIEW

Let’s start by reviewing what LTI is.

LTI REVIEW

— LTI stands for Learning Tools Interoperability.

LTI is an acronym that stands for “Learning Tools Interoperability”. Unlike a lot of technological names, it does actually tell you quite a bit about what it’s all about.

LTI REVIEW

- LTI stands for Learning Tools Interoperability.
- LTI describes how learning platforms can integrate with learning tools.

Generally speaking, LTI is a standard that describes how learning platforms - in our case, that's the edX platform - can seamlessly integrate with learning tools. This seamless integration is done through a browser based launch from a platform to a tool.

Just to give you a real world example of an LTI launch, imagine a learner going to a Coding 101 course in the platform. They visit the course content and are launched to a Jupyter Notebook coding exercise directly from within the courseware.

LTI REVIEW

- LTI stands for Learning Tools Interoperability.
- LTI describes how learning platforms can integrate with learning tools.
- LTI can greatly enrich a learning platform's offerings.

And that's what's so powerful about it. An LTI launch is a way to deeply enrich a platform's content offerings and capabilities without the need for bespoke integrations of custom features built in-house.

LTI REVIEW

- LTI stands for Learning Tools Interoperability.
- LTI describes how learning platforms can integrate with learning tools.
- LTI can greatly enrich a learning platform's offerings.
- LTI is authored by 1EdTech.

LTI is authored by an organization called 1EdTech. 1EdTech has authored a number of versions of the LTI standard, including LTI 1.1 and LTI 1.3.

LTI REVIEW

- LTI stands for Learning Tools Interoperability.
- LTI describes how learning platforms can integrate with learning tools.
- LTI can greatly enrich a learning platform's offerings.
- LTI is authored by 1EdTech.
- Today, we'll be discussing the basic LTI 1.3 launch.

Today, we'll be talking about LTI 1.3 specifically, because it's the only currently supported version of LTI by 1EdTech. We also will not be talking about LTI Advantage Services today.

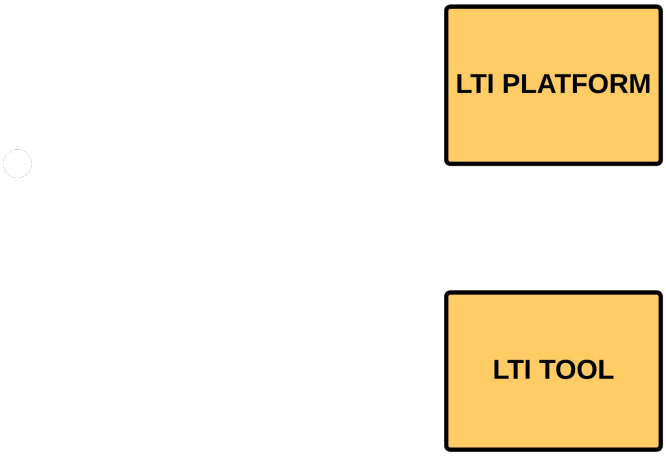
Now that we understand what LTI is and why it's useful, let's take a brief look at how it actually works.

LTI LAUNCH: INTEGRATION

Before an LTI launch ever occurs in the browser, an LTI integration must be established.

LTI LAUNCH: INTEGRATION

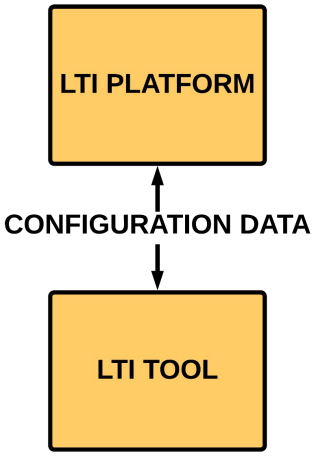
- LTI integrations are created between platforms and tools.



The LTI integration is created between a LTI platform and an LTI tool.

LTI LAUNCH: INTEGRATION

- LTI integrations are created by exchanging configuration data between platforms and tools.



12

In order for an LTI integration to be established, LTI configuration data must be exchanged between the platform and the tool. The set of configuration data is described by the standard, but it includes data like a client ID for the tool, a public key for the tool, various URLs for the LTI launch endpoints, etc.

How this data is exchanged between the platform and tool are outside the scope of the standard.

LTI LAUNCH: INTEGRATION

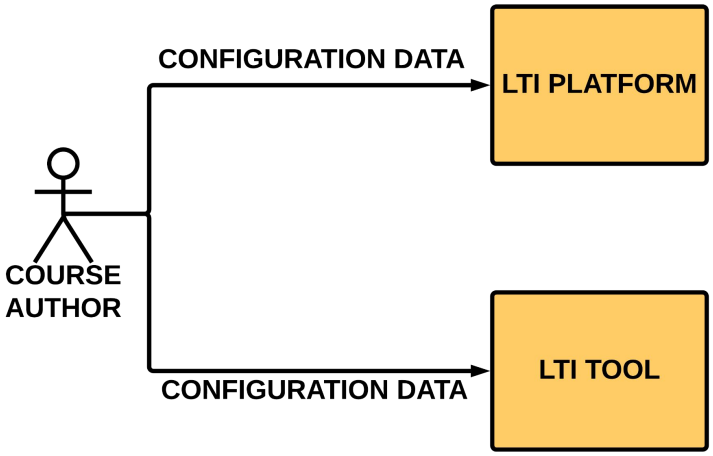
- On edX, the configuration step is done by a course author.

13

But, on the edX platform, this configuration exchange is done by the course author.

LTI LAUNCH: INTEGRATION

- The course author manually exchanges configuration data between the platform and the tool.



14

The course author creates the LTI integration in the platform and in the tool. They manually exchange the LTI configuration data by entering the values in the platform and in the tool. The edX platform stores this configuration data. On the edX platform, the course author does this through Studio by authoring LTI components.

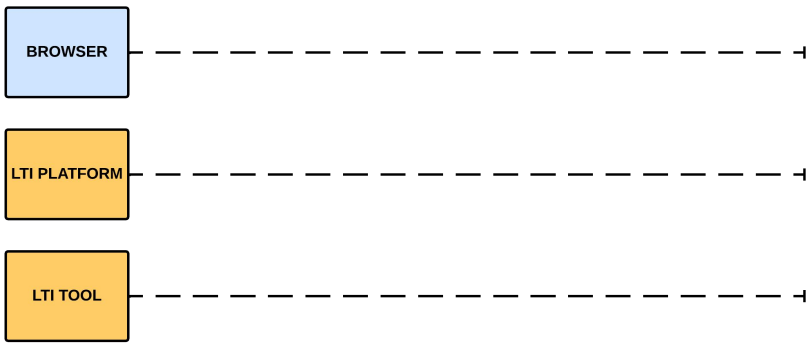
This is an important step in the LTI integration process that must occur before an LTI launch every occurs.

LTI LAUNCH: BROWSER LAUNCH

Now, we're ready to do an LTI browser based launch. We're going to do an abridged look at the LTI launch. We're going to touch on the elements of an LTI launch that are key to understanding why our LTI implementation was coupled to the edX platform.

LTI LAUNCH: BROWSER LAUNCH

- The browser, platform, and tool are key players in an LTI 1.3 launch.

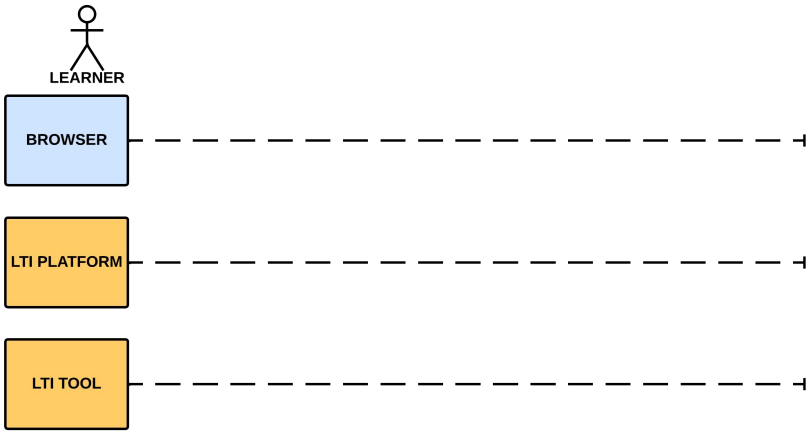


16

Here we have the key players of an LTI launch. Naturally, we have the browser, because this is a browser based launch. And we have the learning platform and the learning tool, which are integrated together through the LTI configuration.

LTI LAUNCH: BROWSER LAUNCH

- A learners clicks an LTI launch link to start the LTI launch.



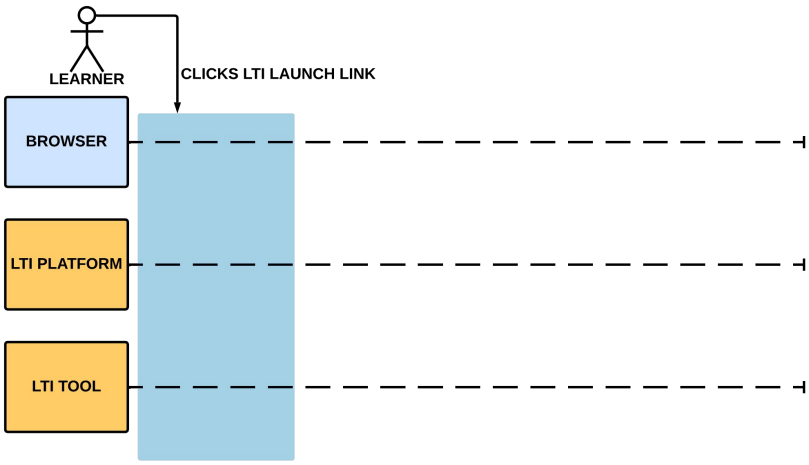
17

Of course, no browser based flow is going to make much sense without a user. Here, we have an edX learner who'd like to launch into Jupyter Notebook through the platform in their Coding 101 course.

First, the learner visits the edX platform. The edX platform renders an LTI launch link. When the learner clicks this link, the LTI launch flow will begin in the browser.

LTI LAUNCH: BROWSER LAUNCH

- The LTI launch begins when the learner clicks the LTI launch link.

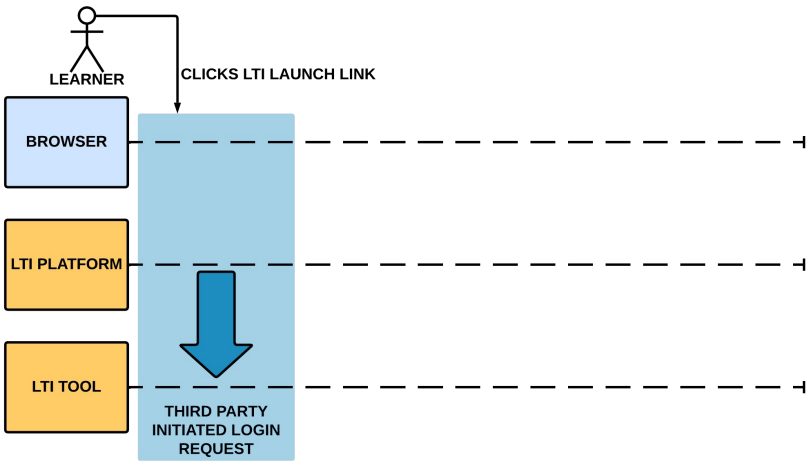


18

The learner clicks the LTI launch link, and the LTI launch flow begins.

LTI LAUNCH: BROWSER LAUNCH

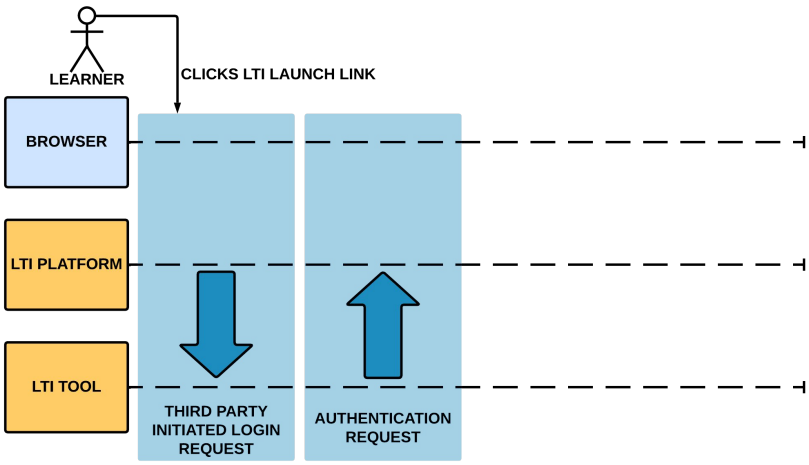
- This is the Third Party Initiated Login Request.
- It informs the tool that an LTI launch has begun and prompts it to make the Authentication Request.



The platform generates a request to the tool via the browser called the Third Party Initiated Login request. This is the platform's way of alerting the tool that an LTI launch is going to begin soon. It also prompts the tool to make the subsequent request.

LTI LAUNCH: BROWSER LAUNCH

- This is the Authentication Request.
- The tool is requesting the the platform authenticate the learner.



20

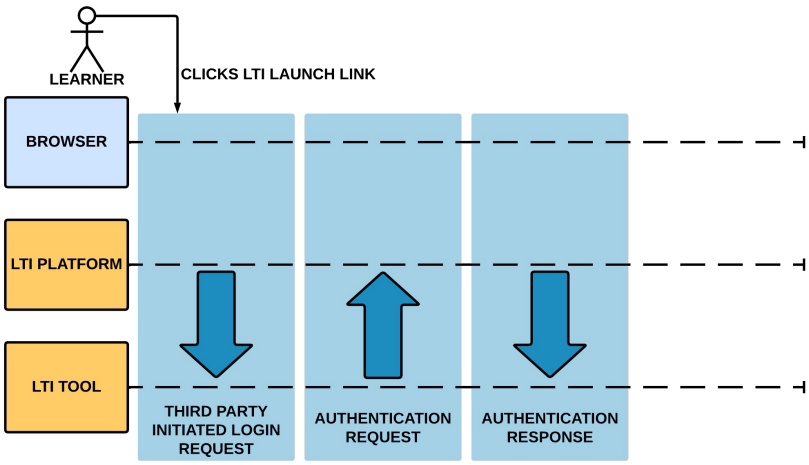
The subsequent request is made by the tool to the platform. The purpose of this request is to ask the platform to authenticate the learner.

For this reason, this request is called the authentication request.

In order for the learner to be able to access tools seamlessly, tools rely on the authorization server, which is the platform in this example, to authenticate the learner. This avoids the needs for learners to maintain separate sets of credentials for each LTI tool. By virtue of being authenticated with the platform, the learner gets an SSO-like experience with tools.

LTI LAUNCH: BROWSER LAUNCH

- This is the Authentication Response.
- This is the actual LTI launch and contains the launch message.
- The message has contextual data.



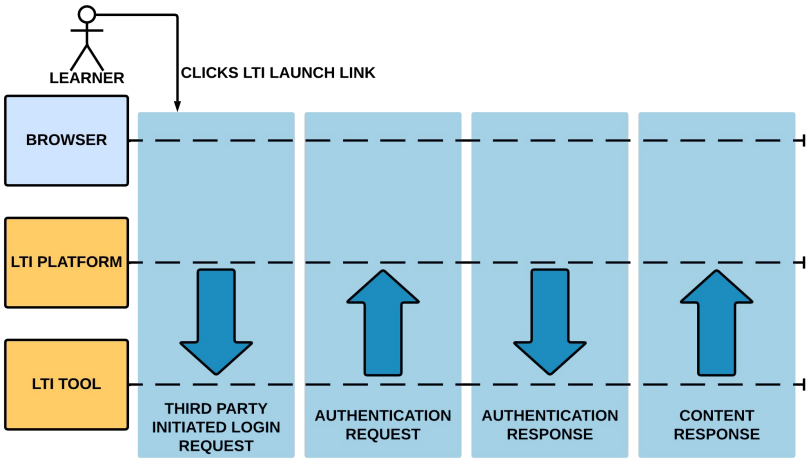
21

Naturally, the platform sends a response to this authentication request to the tool.

This is called the authentication response. It's also the actual LTI launch. In this request, the platform sends a JSON Web Token, or JWT, that contains the learner's identity information as well as other contextual information about the launch. Contextual information include the user's email address, the role of the user on the platform, the context in which the launch is occurring, and so on. This JWT is called the ID token.

LTI LAUNCH: BROWSER LAUNCH

- This is the Content Response.
- The tool is launched to from the platform.



22

And lastly, once the tool receives the LTI launch message, it's ready to return its contents to the platform. It does so by issuing a redirect to its content. The browser does the redirect, and the tool is displayed within the course content. This is called the content response.

There are a lot of boxes and arrows here, and it's not so important that you understand every request and response. There are a few key points we need to take away from understanding how the LTI launch works.

LTI LAUNCH: KEY POINTS

What are the key points?

LTI LAUNCH: KEY POINTS

1. **CONFIGURATION DATA** must be exchanged between platform and tool out-of-band before a launch.

First, we need to understand that an LTI integration must be established before an LTI launch ever occurs in the browser. Configuration data must be exchanged between the platform and the tool during this step.

LTI LAUNCH: KEY POINTS

1. **CONFIGURATION DATA** must be exchanged between platform and tool out-of-band before a launch.
2. The LTI launch occurs in the **BROWSER** via two **REQUEST-RESPONSE** cycles or **LEGS** of the LTI flow.

Next, it's important to understand that the LTI launch occurs via the browser. The platform makes a series of requests to the tool and vice-versa.

LTI LAUNCH: KEY POINTS

1. **CONFIGURATION DATA** must be exchanged between platform and tool out-of-band before a launch.
2. The LTI launch occurs in the **BROWSER** via two **REQUEST-RESPONSE** cycles or **LEGS** of the LTI flow.
3. The platform must send **CONTEXTUAL DATA** to the tool in the LTI launch.

Lastly, in the actual LTI launch, the platform must send important data about the LTI launch and the context in which it's occurring. This is what we decided to call contextual data. This is the information contained in the ID token that's sent in the LTI launch.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

Now, we've reviewed what LTI is, why it's important, and how it works. But if LTI is such a powerful extensibility mechanism, then what is the problem?

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

In order to understand the problem with our LTI implementation, we need to use our imagination and pretend it's March 30th, 2022. You're on the Cosmonauts team at 2U, and you want to improve your team's implementation of proctoring. Currently, your proctoring integrations are custom and bespoke, and it's hard to add new ones, let alone support the old ones. You want to use LTI to launch to proctoring vendors in the courseware instead. But you quickly run into a problem. Let's take a look at what that is.

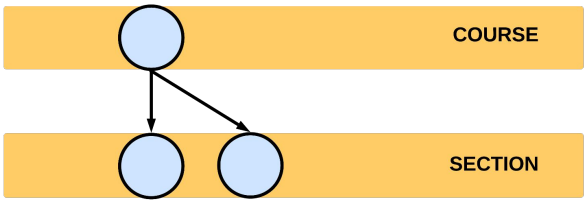
LTI ON EDX

In order to understand the conflict between LTI and proctoring, we need to take a look at how LTI is supported on the edX platform. Let's take a look at a sample representation of a course on edX platform as a tree to better understand this.

LTI ON EDX

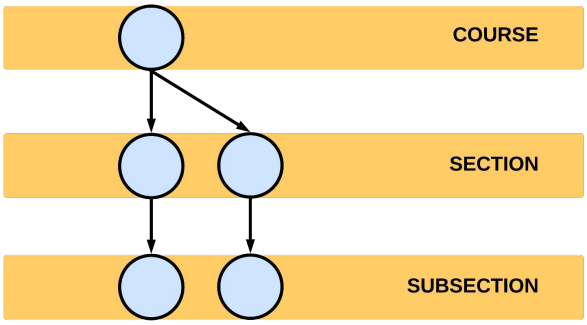
At the root of the course tree, we have the course block.

LTI ON EDX



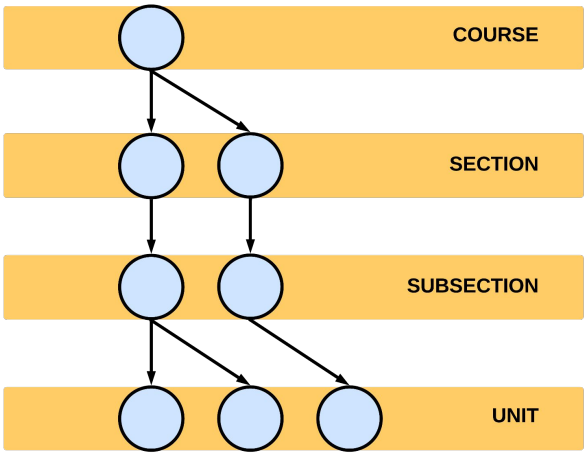
The course block has section blocks as children.

LTI ON EDX



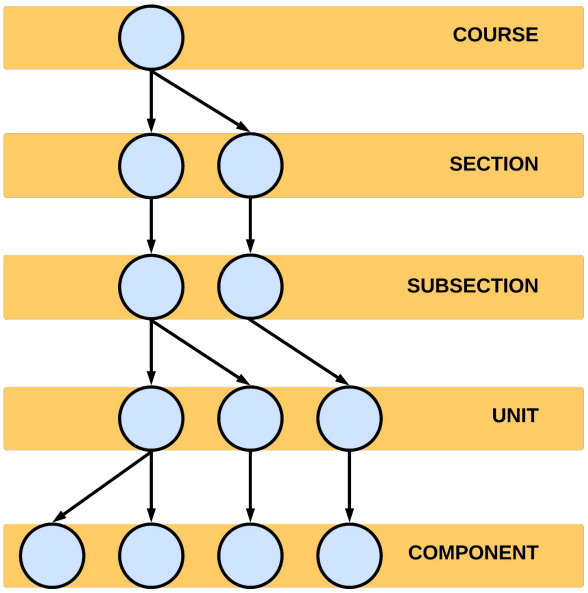
Section blocks have subsection blocks as children.

LTI ON EDX



Subsection blocks have unit blocks as children.

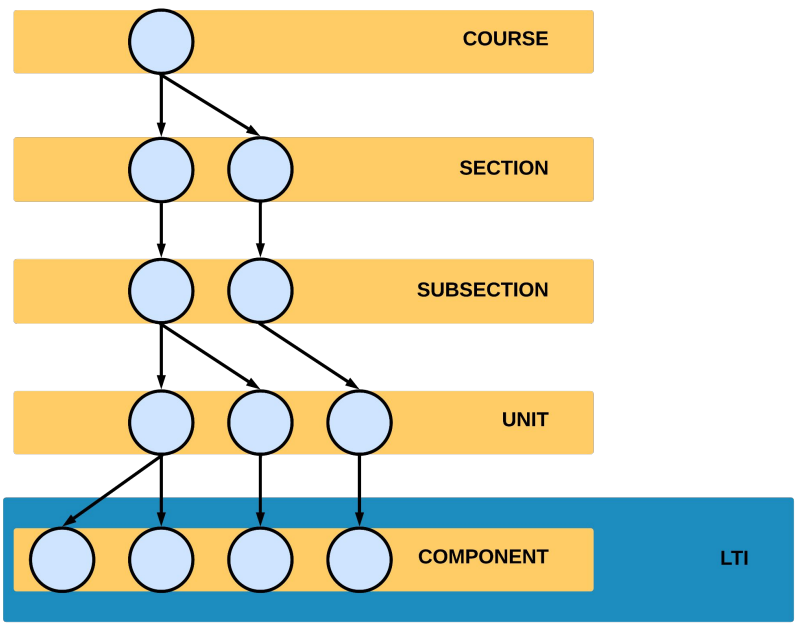
LTI ON EDX



And, finally, unit blocks have component blocks as children. Why is this important?

LTI ON EDX

- LTI launches occur at the component level.

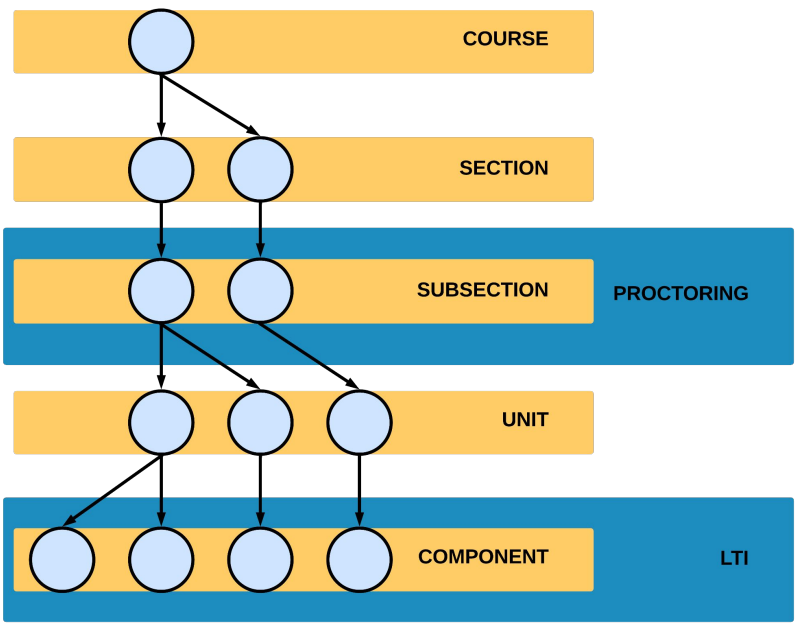


35

Well, it turned out that LTI launches could only be done at the component level. This is because LTI is implemented as an XBlock. An LTI launch can only be done at the component level in the courseware.

LTI ON EDX

- LTI launches occur at the component level.
- Proctoring occurs at the subsection level.

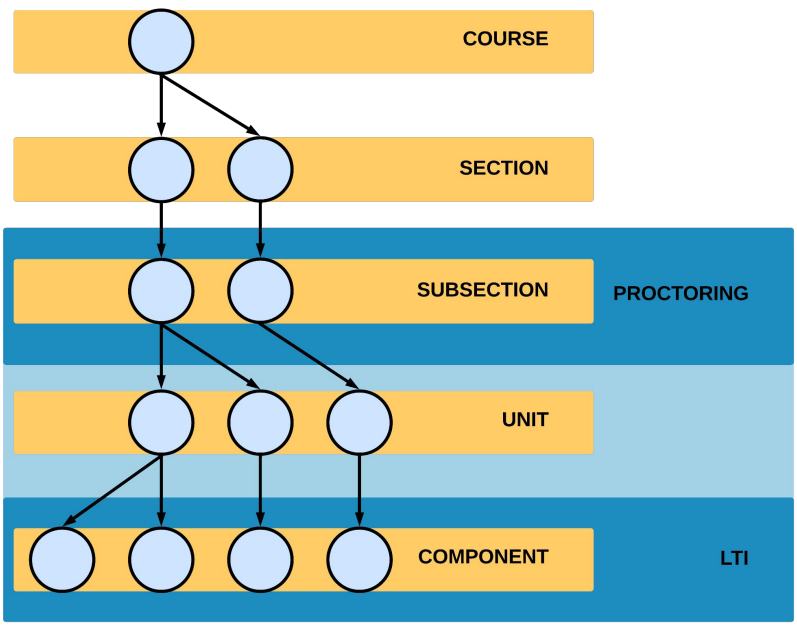


36

Unlike LTI, proctoring is set at the subsection level. It's a property of a subsection.

LTI ON EDX

- LTI launches occur at the component level.
- Proctoring occurs at the subsection level.
- LTI launches cannot occur at the subsection level.



37

This means that all descendants of the proctored subsection are a part of the proctored exam.

Unfortunately, this meant we couldn't use LTI for proctoring, because the proctoring experience occurs at the subsection level, and our implementation of LTI was only possible at the component level.

This brings us to our overall problem statement, which applies to more than just proctoring.

PROBLEM STATEMENT

The implementation of LTI in the XBlock made it impossible to do an LTI 1.3 launch from outside the courseware.

This is our overall problem statement, and this problem is the one we endeavoured to solve. We wanted to make it possible to do a basic LTI 1.3 launch from any Django application and from outside the courseware.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

Now that we've understood the problem we were facing,

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

let's talk about what we did about it.

I'm going to walk us through an overview of the architecture of our LTI implementation describe what our strategy was at a high level before diving into more of the details.

HIGH-LEVEL STRUCTURE

Let’s start by talking about the structure of our LTI implementation and its relationship to the edX platform.

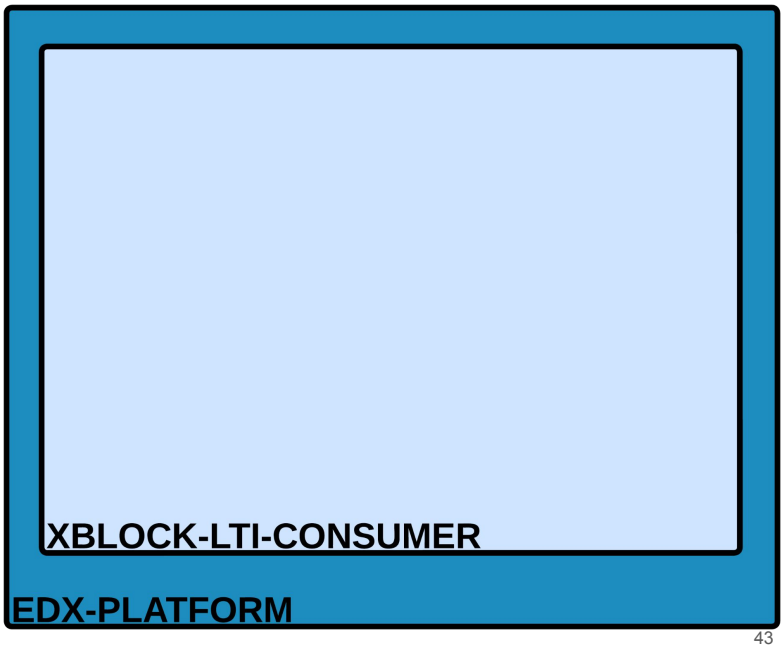
HIGH-LEVEL STRUCTURE

- This is the edX platform.

Here is the edx-platform.

HIGH-LEVEL STRUCTURE

- The LTI library is installed into the edX platform.

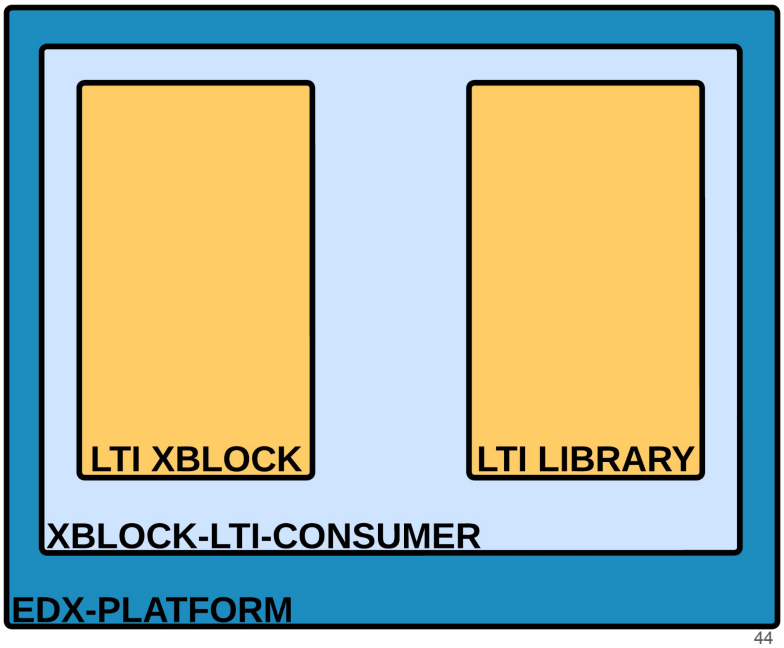


And this is the xblock-lti-consumer library.

The xblock-lti-consumer is a library that contains our LTI implementation. The library is installed into the edX platform.

HIGH-LEVEL STRUCTURE

- The LTI library contains the LTI XBlock and all the supporting LTI code.



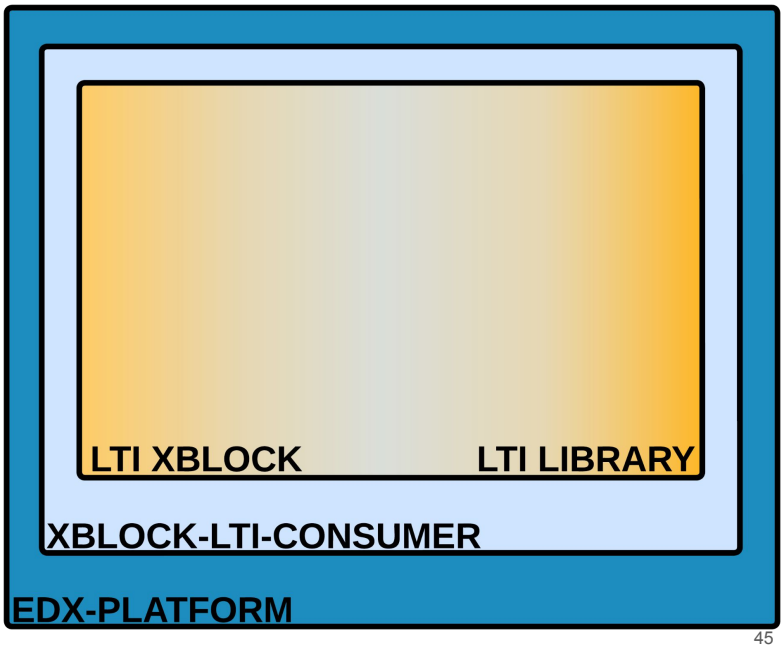
44

The xblock-lti-consumer library contains an XBlock, called LtiConsumerXBlock, and all of the supporting code that makes LTI possible. This XBlock is installed into the platform and made available as any other XBlock is.

You might imagine that it would be possible to, say, just install the LTI library anywhere it could be useful, leveraging that supporting code in other contexts beyond the XBlock runtime. Unfortunately, we found that that wasn't possible.

HIGH-LEVEL STRUCTURE

- There weren't clear boundaries between the LTI XBlock and the supporting LTI code.



45

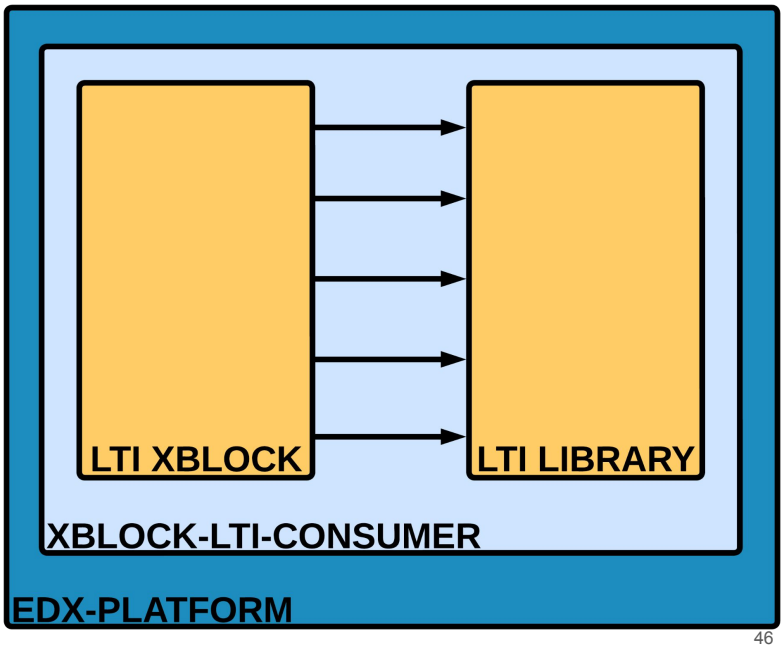
This is a more accurate representation of the library and of the relationship between the LtiConsumerXBlock and the LTI library components.

The LTI XBlock and the LTI library had grown together, and it became difficult to see clear boundaries.

This made it impossible to do an LTI 1.3 launch outside the XBlock runtime. In order to enable our proctoring changes, we sought to break them up.

HIGH-LEVEL STRUCTURE

- Our goal was for the LTI XBlock to depend on the LTI library.
- This way, the LTI library could be installed elsewhere.



46

This was our aspirational design. We wanted an independent core LTI library that various contexts could install and make use of.

In this design, the XBlock now depends on the LTI library, but the LTI library does not depend on the XBlock. This frees the library up to be separated into a separate repository and installed as needed by other contexts.

Of course, this is aspirational. And we didn't completely get to this point. But we did get close enough to enable basic LTI 1.3 launches from anywhere.

I also want to take a moment to mention that a lot of this work built on OpenCraft's heavy investment in the repository long before we become involved. They did a lot of work on LTI, and the work I'm presenting to you today would not be possible without their involvement and expertise. I want to be sure to give credit where it's due.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

That's a look at our overall vision and approach for the LTI library, but before we move on to a closer look at the details,

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

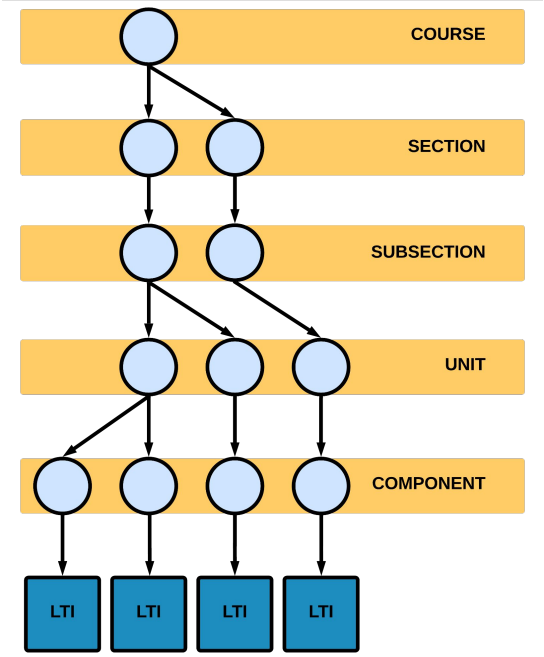
I want to take a moment to reflect on the impact of these changes. Of course, you can appreciate the impact this has on LTI proctoring project, but it's important to understand how these change impact the entire platform beyond just proctoring. Here are a few ways that a decoupled LTI greatly enhances our platform capabilities.

IMPACT:
LTI REUSE

First, there’s LTI reuse.

IMPACT: LTI REUSE

- Each LTI component is associated with a single LTI integration.



50

Here is another look at our course tree structure from before.

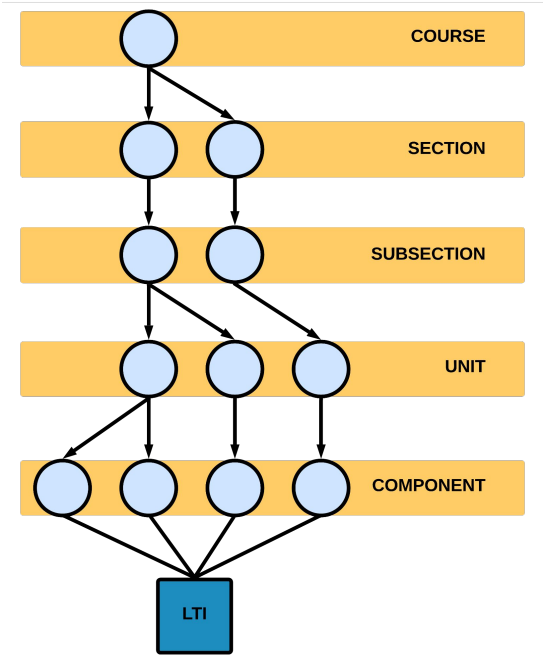
Notice that each LTI component has a one-to-one relationship with an LTI integration. Because LTI is implemented as an XBlock, the LTI configuration information is stored in the modulestore. This means that each XBlock has its own LTI configuration.

The problem with this is that course authors will often want to place the same LTI tool multiple times in a course. Theoretically, the LTI integrations could be identical, but because of our LTI implementation, course authors would have to painstakingly create LTI integrations in Studio and in the tool for each placement of the tool.

That has been our biggest point of feedback as more course teams are willing to experiment with LTI 1.3 It's a major impediment to LTI 1.3 adoption on the platform.

IMPACT: LTI REUSE

- LTI components can share a single LTI integration.



51

If LTI is decoupled from the XBlock, and LTI configuration data no longer has to be stored in the modulestore, LTI integrations can be shared amongst many LTI components.

This makes authoring LTI 1.3 components a lot simpler. In fact, we are currently working on early plans to investigate LTI configuration reuse between XBlocks that become possible as a result of this work.

Of course, this improvement is largely a consequence of a change to our implementation of LTI. It's not a benefit that we get as a result of using LTI in more places. Let's take a look at some course authoring improvements we can get as a result of doing LTI launches from more contexts.

IMPACT: STUDIO

- A decoupled LTI enables LTI integrations from other parts of the platform, like Studio.

Here is the course outline in Studio. Course authors design their courses here. Designing a course with accessibility in mind is an important part of having equitable and impactful content. Imagine if we did an LTI launch to a content scanning tool that would scan course content for accessibility best practices. This would enable course teams to design more accessible course content.

IMPACT: LMS

- A decoupled LTI enables LTI integrations from other parts of the platform, like the LMS.

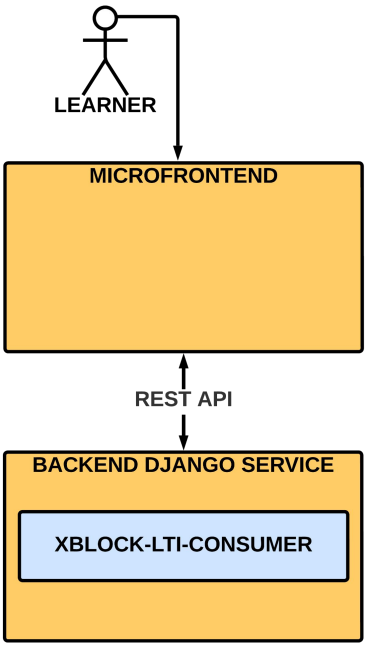
On the learning side, LTI can come in handy too. This is the discussion forums. There's a lively discussion going on. We could integrate with an LTI tool to implement talk-to-text on the page or to read the discussion forum aloud to a learner. This makes participating in a course easier.

IMPACT: DJANGOAPP

Let’s end with what I think is the biggest impact, and that’s that LTI launches can be done from any Djangoapp. Let’s take a look at a simple Djangoapp to see how this could work.

IMPACT: DJANGOAPP

- A decoupled LTI enables LTI launches from a Djangoapp.



55

This is a very basic look at how LTI could be done from a Djangoapp. This is the model we're using for our proctoring project.

We've got the microfrontendend serving up content in the browser, and we've got the backend Django service powering it.

The xblock-lti-consumer library is installed into the Django service, and it's providing the implementation of LTI. In particular, it can generate the LTI launch link and handle all the backend request handling for the two legs of the LTI launch flow.

When the learner interacts with the MFE in the browser, it calls to the backend Django service, which generates and returns an LTI launch link to the MFE. The launch link is rendered in the browser. When the learner clicks it, the LTI launch starts, and two legs of the LTI launch flow are handled by the LTI library in the Django service.

These are some simple examples of how LTI could be used outside of the XBlock runtime to enhance the platform. Although we may never implement the LTI tool ideas in Studio and the LMS, I'm still really excited about this work. And that's because the fact that they're possibilities now is a big change. We're no longer limited by our implementation of LTI. We have the entire marketplace of LTI tools at our disposal.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

This gives us an appreciation for the importance of this work and why it goes beyond just proctoring.

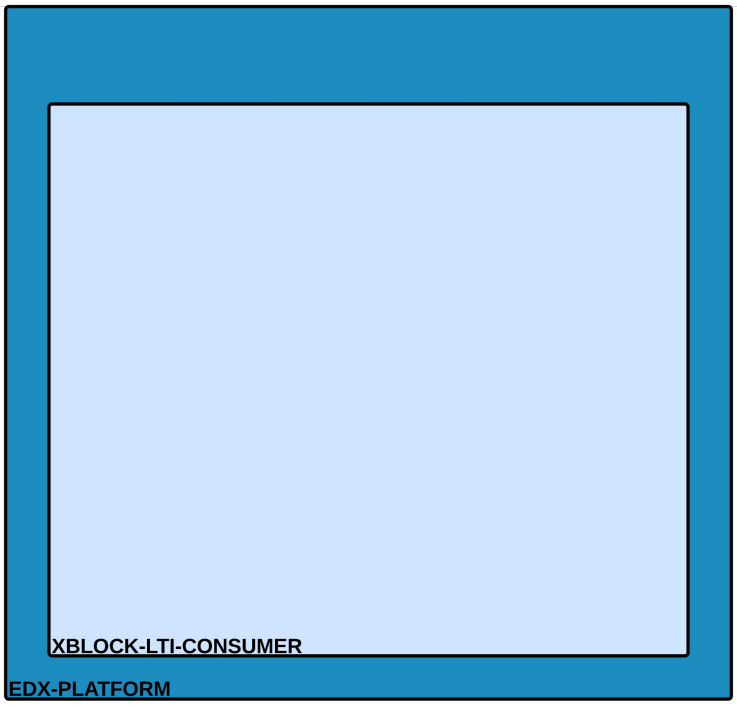
AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

Let's dive into how we actually did the decoupling. We're going to look at the same problem and the same approach but under a microscope.

LTI DESIGN

- The LTI library is installed into the edX platform.

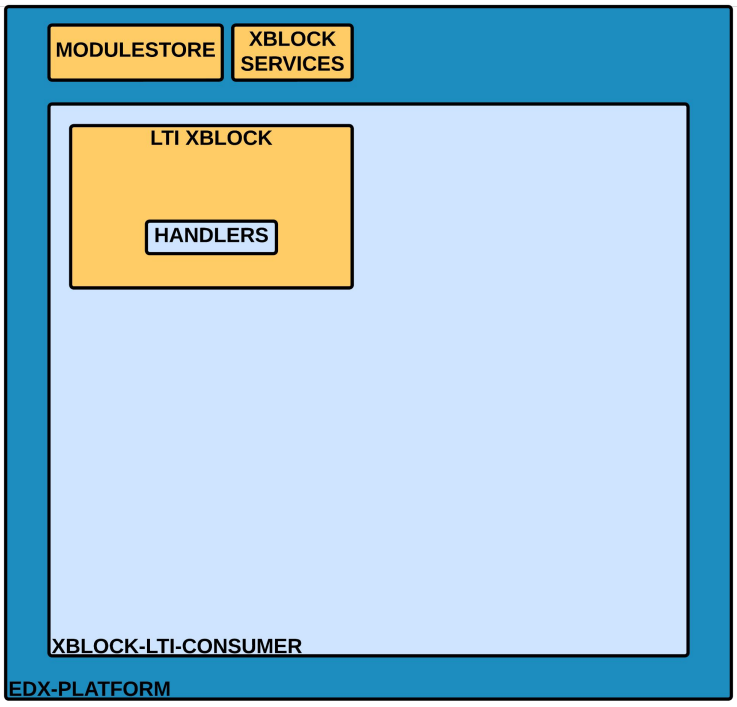


Again, this is the edX platform. The xblock-lti-consumer library is installed into it.

Let’s take a look at the key components of the edX platform and the xblock-lti-consumer with respect to LTI and understand the relationships amongst them.

LTI DESIGN

- LTI XBlock has handlers.
- Handlers implement LTI launch flow.
- LTI XBlock runs in XBlock runtime with modulestore and XBlock services.



59

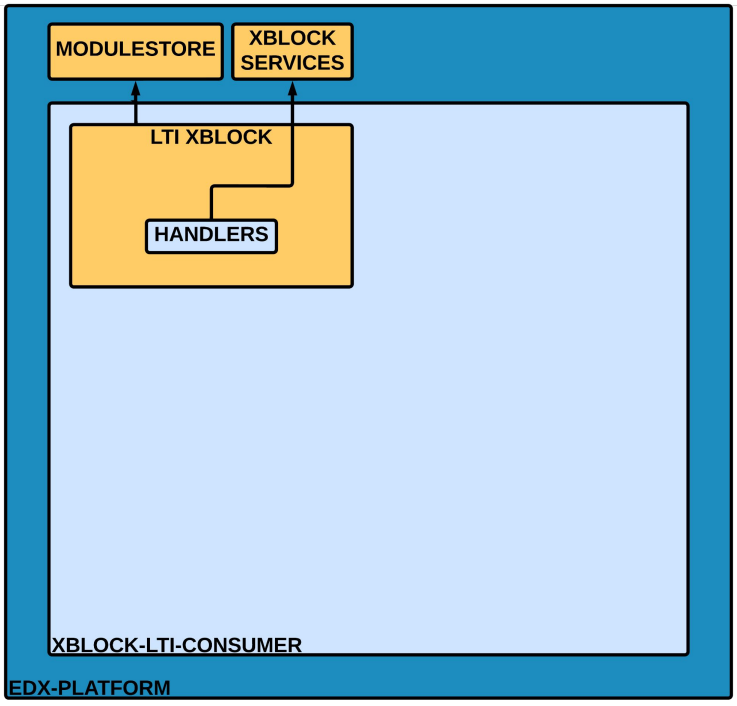
In the xblock-lti-consumer library, we have an LTI XBlock that implements LTI.

The XBlock has Python methods that implement the server side of the XBlock. They are are called “handlers”, and they “handle” AJAX calls made by the learner’s browser. As you’ll recall, LTI is a browser based launch, so these handlers implement the requests made to the platform and return responses. In our case, the handlers implement the request handling for the requests the tool directs to the platform.

Unsurprisingly, there’s the modulestore. The modulestore is where course content is stored, including LTI components. We also have XBlock services that the XBlock relies on.

LTI DESIGN

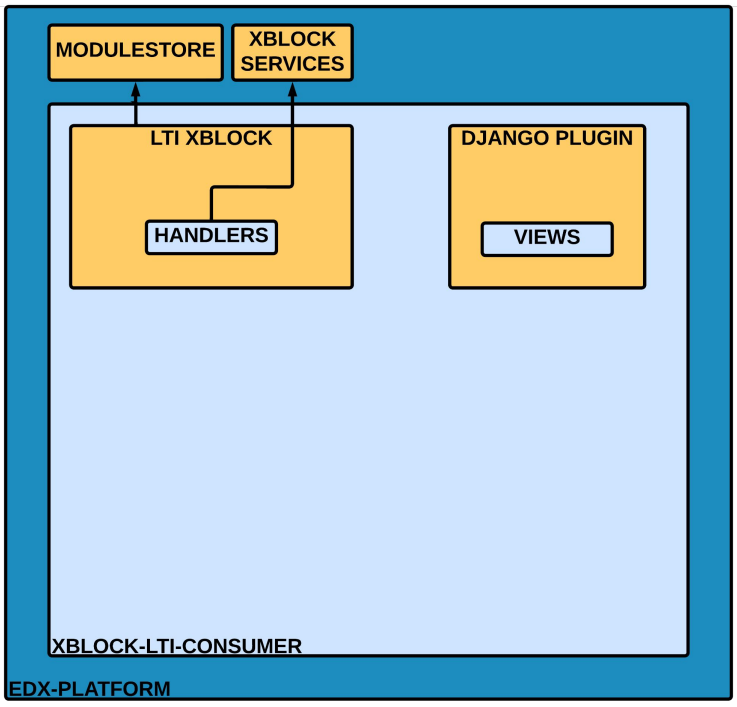
- LTI XBlock relies on modulestore and XBlock services.



The LTI XBlock has a dependency on the modulestore and on XBlock services. It uses the XBlock services to collect contextual data. For example, it may need a user's email address to include in an LTI launch, which it will get from the XBlock user service.

LTI DESIGN

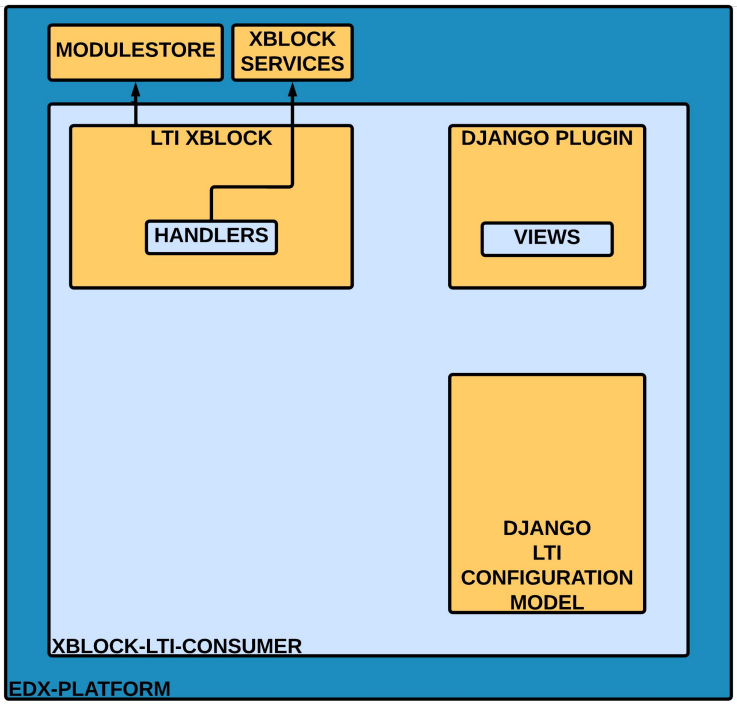
- Django plugin supplies views to installing Djangoapp.
- Django plugin views implement LTI launch flow.



Then, we have a Django plugin. The Django plugin has a few views that implement the request handling for the requests the tool directs to the platform. We'll talk about the relationship between the LTI XBlock handlers and the Django plugin views in a moment.

LTI DESIGN

- Django model stores small amount of LTI configuration data.

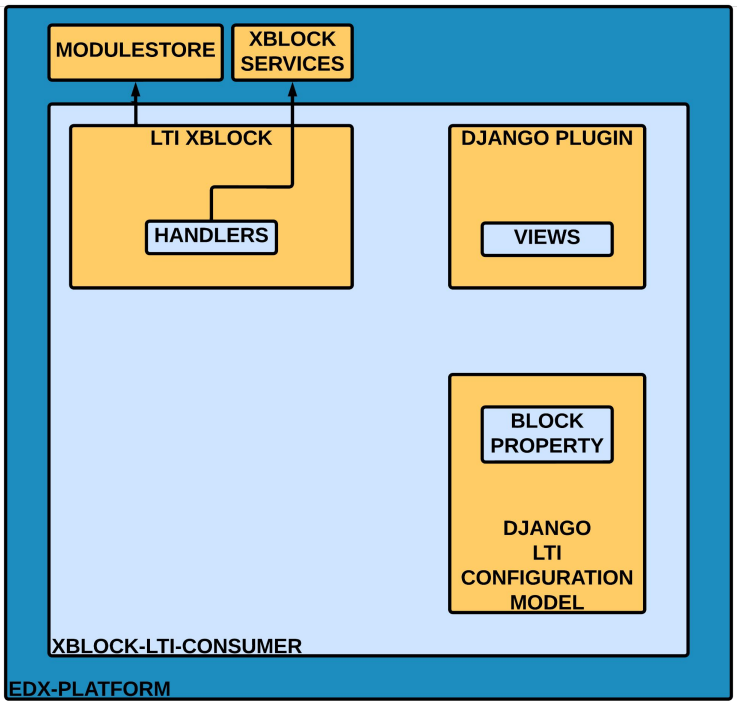


We also have a LtiConfiguration Django model. This Django model stores a small amount of LTI configuration data. Although the LTI XBlock stores the majority of LTI related configuration, the LtiConfiguration model stores configuration data that is auto-generated by the platform. For example, a client ID for the tool and the platform's public-private key pair is auto-generated by the platform and stored in this model.

Of course, this now means that the LTI configuration data for a particular LTI integration is now split between the LtiConfiguration model and the LTI XBlock.

LTI DESIGN

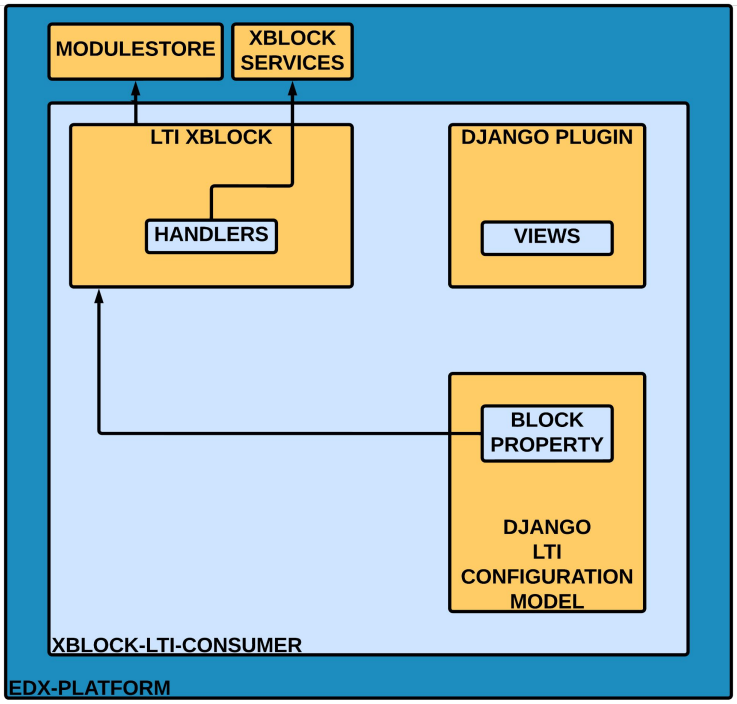
- Django model has a block property.



That's where the block property of the LtiConfiguration model comes in.

LTI DESIGN

- Block property of Django model ties XBlock and Django model together.



64

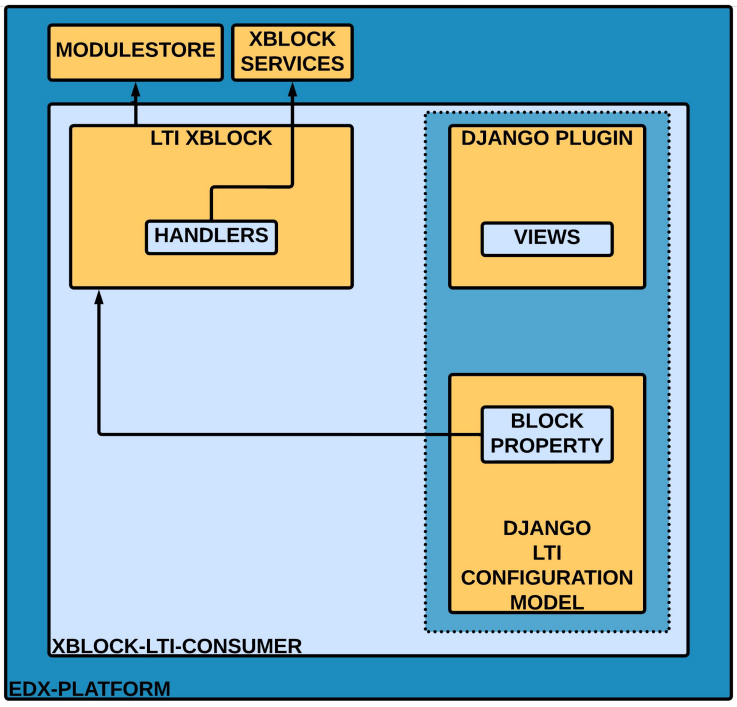
In order to connect the two sources of LtiConfiguration, the block property holds a reference to a block descriptor that has been loaded out of the modulestore. This allows the LtiConfiguration model to have access to the full set of LTI configuration data.

Now, these are the main dependencies and relationships between these components. Our approach to decoupling the LTI implementation from the XBlock was to continue this exercise of enumerating the relationships between the XBlock and everything else LTI related. Using this process, we identified three key coupling points. We believed that if we addressed each of these three coupling points, we could decouple the LTI implementation from the XBlock. Let's take at each one and discuss how we went about addressing them.

There are going be small changes to the diagram as we're reviewing these coupling points. I've included a green star icon on the diagram to indicate what part of the diagram we're talking about and what part of the diagram has changed. If you find yourself getting lost in the diagram, just refer to the green star.

LTI DESIGN

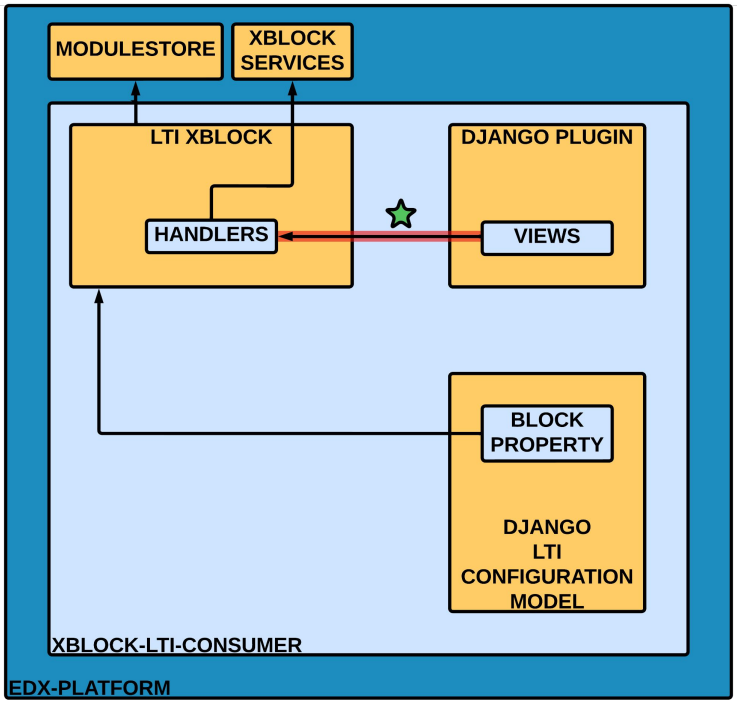
- The Django plugin and model are the part of the LTI library that can be used outside the XBlock runtime.



I want to take a moment to point out that the Django plugin and Django LTI configuration model are a part of the “LTI library” - the parts of the xblock-lti-consumer library that are intended to be usable outside the XBlock runtime. As we walk through the diagram, keep in mind that we want to see the LTI XBlock depend on this library but not vice-versa.

KEY COUPLING POINTS: REQUEST HANDLING

- Django plugin views call XBlock handlers.



Our first dependency is between the Django plugin views and the XBlock handlers. I promised you I'd explain what was going on with the XBlock handlers and the Django plugin views. Well, this plugin was created as a first step toward a decoupled implementation. Like the XBlock handlers, its intention was to contain the views necessary to implement an LTI launch. The plugin was intended to eventually be independent of the XBlock. However, at this point, it was still dependent on the XBlock.

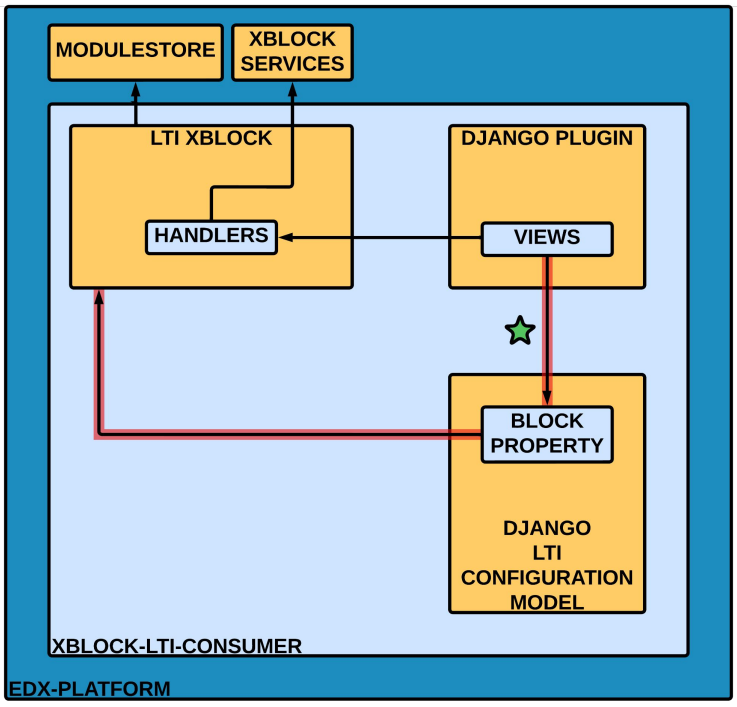
The view for the basic LTI launch simply served as a pass through for the XBlock handler by calling it.

However, the Django view also had a few views that were implemented entirely in the Django plugin. These were mainly views implementing the LTI Advantage Services. Although these views were implemented entirely in the Django plugin, they still needed access to LTI configuration data and LTI contextual data.

This brings us to our next coupling point.

KEY COUPLING POINTS: CONFIGURATION DATA

- Django plugin views access XBlock reference using block property.



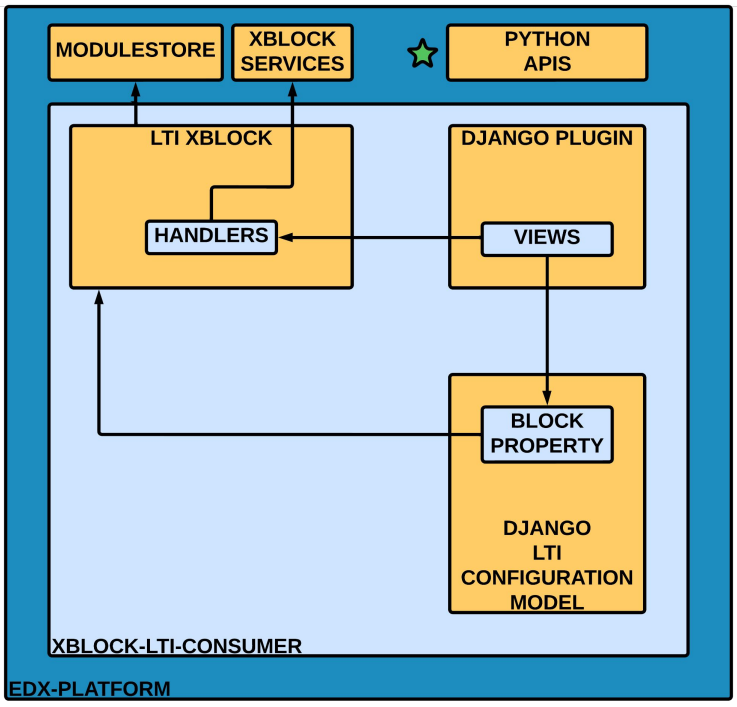
The next coupling point is between the Django plugin views and the block property of the LtiConfiguration model. As I said, the Django plugin needed access to LTI configuration data to implement the LTI launch. The majority of this configuration data was stored in the modulestore, so the Django plugin view access the block property of the model and read the configuration data directly off the XBlock.

This meant that the Django plugin views could not run outside the context of the edX platform.

I also mentioned that the Django plugin views needed access to contextual data. As a reminder, this is data that is sent in the LTI launch - data like the learner's user identifier, the learner's username and email, the learner's role in the course, the name of the course, and so on. How did the Django plugin views get access to this contextual data?

KEY COUPLING POINTS: CONTEXTUAL DATA

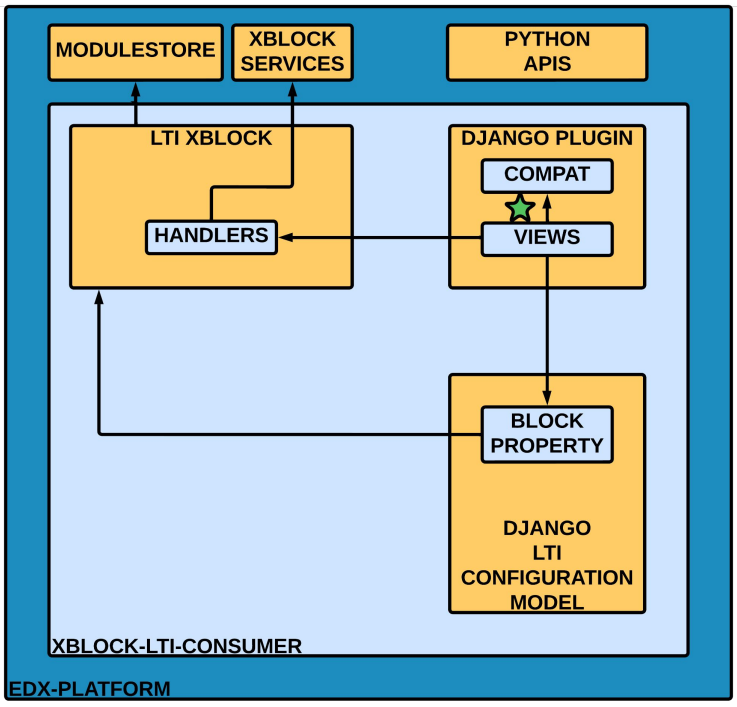
- Django plugin uses platform Python APIs.



Well, they actually called into various Python APIs on the platform.

KEY COUPLING POINTS: CONTEXTUAL DATA

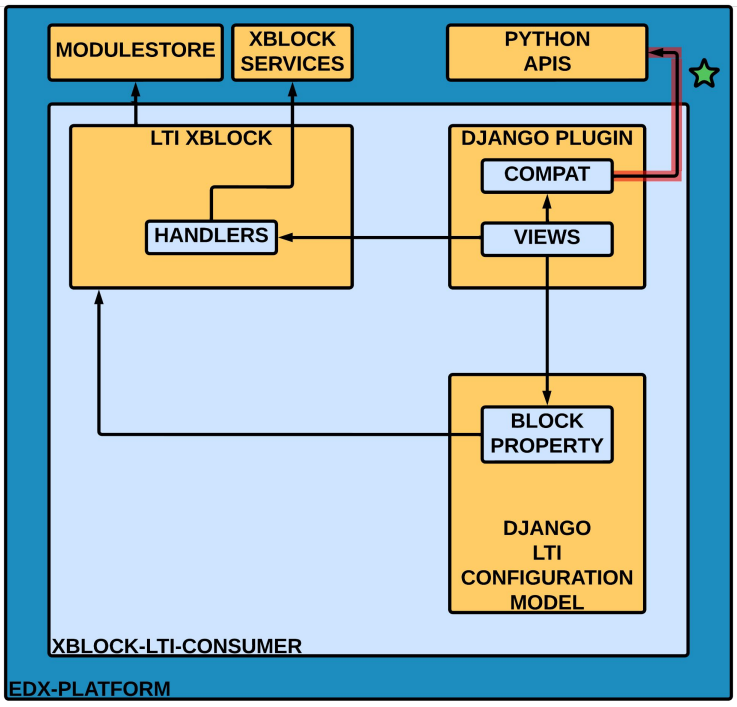
- Django plugin views use a compatibility module.



The Django plugin contained a compatibility module, and the views imported and called functions from this module.

KEY COUPLING POINTS: CONTEXTUAL DATA

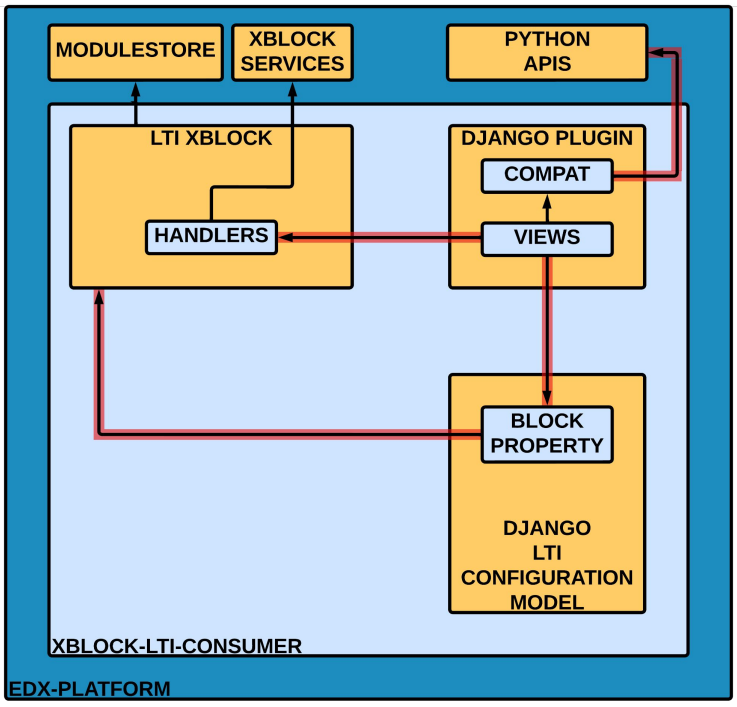
- Compatibility module imports Python APIs from the platform.



The functions of the compatibility module imported various Python API functions from the edX platform and encapsulated them in wrapped functions. For example, the Django plugin views that simply called the corresponding XBlock handlers did so by importing the run_xblock_handler function from the courseware app of the LMS via the compatibility module.

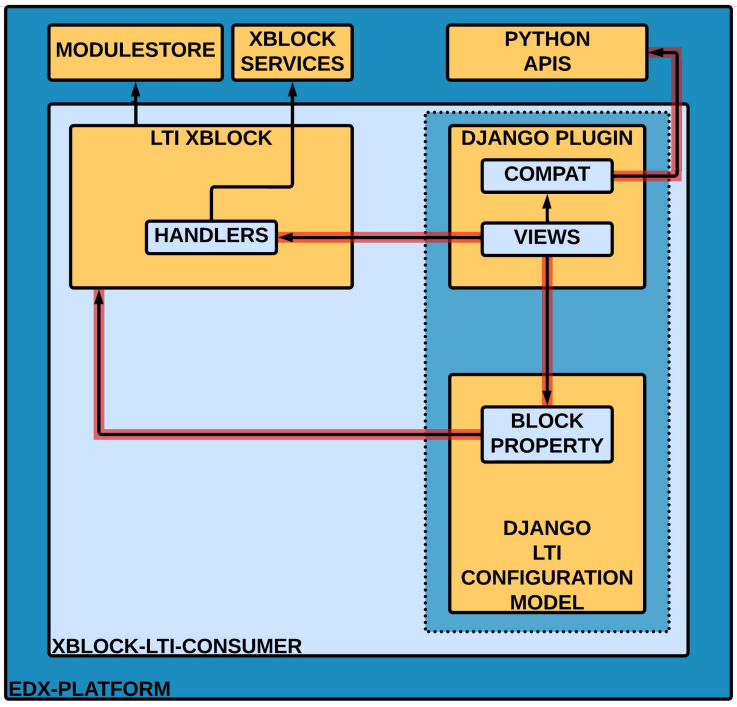
Of course, this meant that these compatibility functions would not return meaningful contextual information unless the Django plugin views were running inside the platform.

KEY COUPLING POINTS: ALL



Here is a look at all three couplings points. As we can see, the direction of the dependencies is from the components of the LTI library to various parts of the edX platform - the XBlock, the XBlock handlers, and the platform APIs.

KEY COUPLING POINTS: ALL



KEY COUPLING POINTS: REVIEW

Let’s review the key coupling points.

KEY COUPLING POINTS: REVIEW

1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.

KEY COUPLING POINTS: REVIEW

1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.
2. The plugin views read **CONFIGURATION DATA** from the XBlock.

KEY COUPLING POINTS: REVIEW

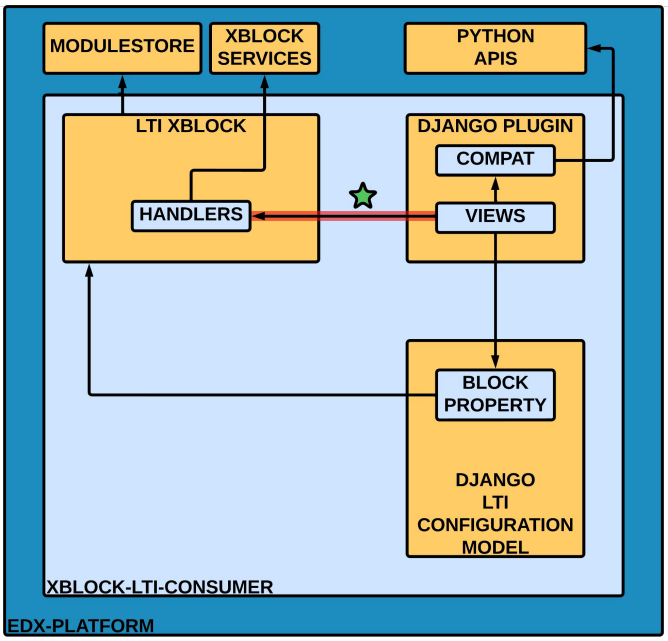
1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.
2. The plugin views read **CONFIGURATION DATA** from the XBlock.
3. The plugin views read **CONTEXTUAL DATA** from the platform.

KEY COUPLING POINTS

1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.
2. The plugin views read **CONFIGURATION DATA** from the XBlock.
3. The plugin views read **CONTEXTUAL DATA** from the platform.

KEY COUPLING POINTS: REQUEST HANDLING

- Django plugin views call XBlock handlers.



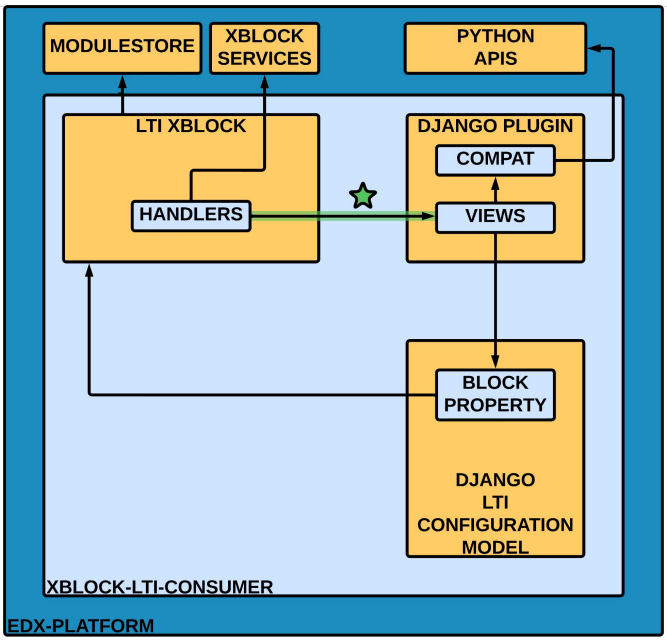
78

Our first dependency is the one between the Django plugin views that the XBlock handlers. In order for these Django plugin views to be useful outside of the XBlock runtime, they'd have to stop calling the XBlock handlers. So, what did we do?

Fortunately, we didn't have to do anything. OpenCraft was working in this repository as we were getting spun up with our LTI for proctoring project. They also had a need for a decoupled LTI, so they were already in the process of addressing this issue. Members of OpenCraft like Giovanni Cimolin da Silva and Arunmozhi Periasamy went through the process of refactoring these Django plugin views. What OpenCraft did was rewrite the XBlock handlers as Django plugin views.

KEY COUPLING POINTS: REQUEST HANDLING

- XBlock handlers call Django plugin views or simply rely on plugin installation.



Now, the dependency looked like this. This is great. We've got a dependency arrow leading from the XBlock to the Django plugin. However, rewriting the request handler implementation as a Django plugin view came at a cost. Although the code moved into a view that can run outside the XBlock runtime, the views still needed access to LTI configuration data.

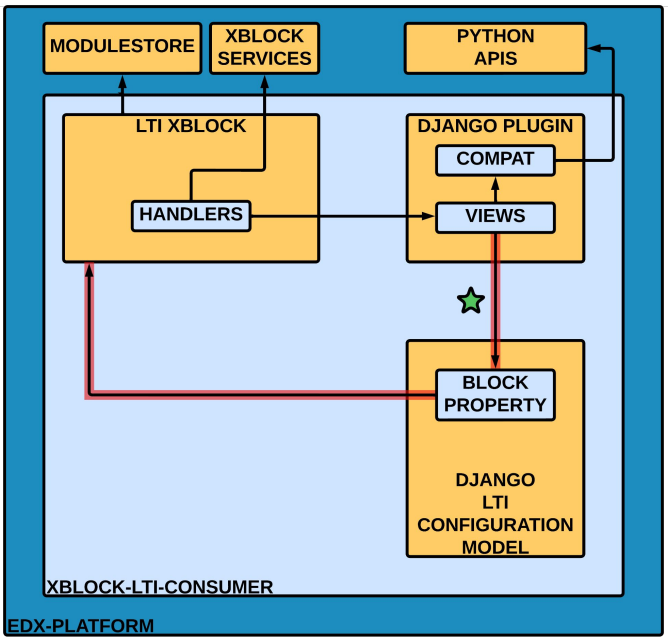
KEY COUPLING POINTS

1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.
2. The plugin views read **CONFIGURATION DATA** from the XBlock.
3. The plugin views read **CONTEXTUAL DATA** from the platform.

This brings us to our second coupling point.

KEY COUPLING POINTS: CONFIGURATION DATA

- Django plugin views access XBlock reference using block property.



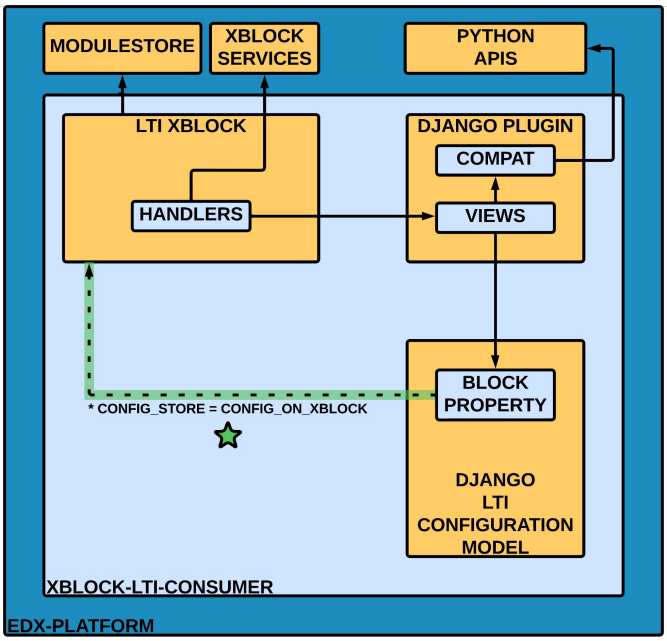
81

Remember, when the Django view is executing, it will need access to LTI configuration data. It does this by reading the data from the XBlock reference that's stored in the block property of the LtiConfiguration property.

The issue here is that the modulestore is the source of truth for the majority of LTI configuration data. We thought, "what if the source of truth for the data could depend on the context the LTI launch is running in?" This brought us to our solution to this problem.

KEY COUPLING POINTS: CONFIGURATION DATA

- Django plugin views use block property depending on config_store field value.



We decided to make the LtiConfiguration model a sort of broker of LTI configuration data. It would define a few different ways that LTI configuration data could be stored and know where to read from. It turns out the LtiConfiguration model already had this feature. The model had a *config_store* field that described where the LTI configuration data was coming from. At the time, there were two options.

1. CONFIG_ON_XBLOCK: This told the model to read the configuration data from the XBlock via the block property.
2. CONFIG_EXTERNAL: This told the model to read from a Django plugin via a filter hook.

What we did is add a third option, CONFIG_ON_DB. This option told the model to read the configuration data from itself. We added all necessary configuration data as fields on the model to support this option.

This would allow a Django application to store all of the configuration data on an instance of the LtiConfiguration model instead of a small amount in the model and the remainder in the modulestore.

That's why this arrow has become dotted and has been annotated with the config_store = CONFIG_ON_XBLOCK label. Although the direction of the arrow has not changed, the coupling has become loose and conditional on that field.

[However, the CONFIG_ON_DB option doesn't work for the XBlock. It made sense for

the source of truth for LTI configuration data to remain on the XBlock for the time being. There were over 55,000 LTI blocks on the platform, and it was out of scope for us to consider making the LtiConfiguration model the source of truth for the XBlock too. Although this is not ideal, it was sufficient for decoupling the data storage.]

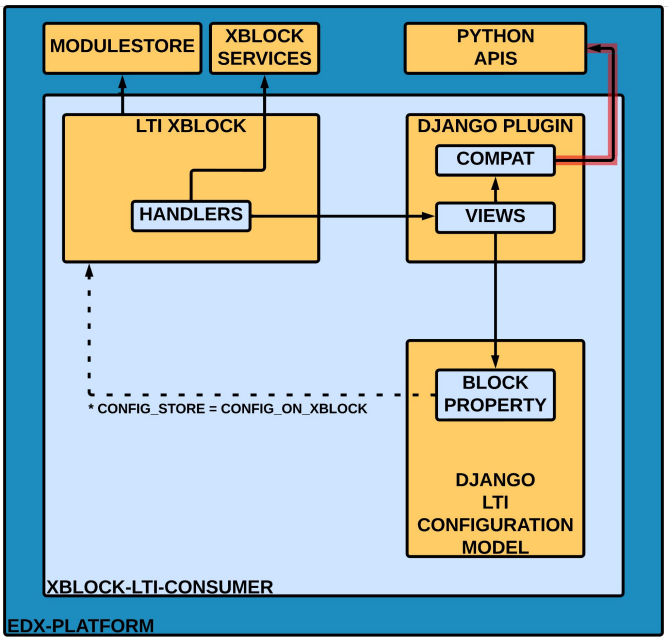
KEY COUPLING POINTS

1. The plugin views implement **REQUEST HANDLING** by calling XBlock handlers.
2. The plugin views read **CONFIGURATION DATA** from the XBlock.
3. The plugin views read **CONTEXTUAL DATA** from the platform.

Lastly, we need to tackle the issue of contextual data.

KEY COUPLING POINTS: CONTEXTUAL DATA

- Compatibility module imports Python APIs from the platform.



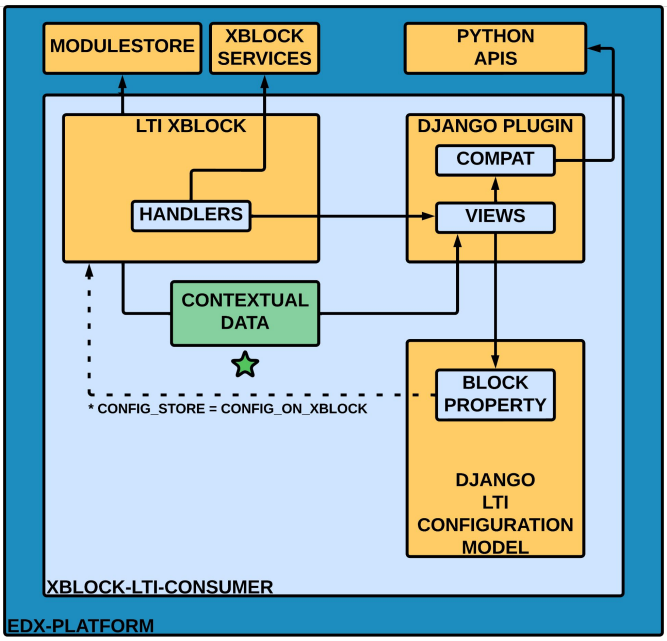
The relationship between the Django plugin and the platform APIs exists because the Django plugin needs access to contextual data that is defined by the platform. These data may include things like a user email, the user's roles, the course the launch is occurring in, and so on.

The issue is that how a piece of contextual data, like a user's roles, is defined and retrieved depends on the context that an LTI launch is occurring in. Our goal was to enable the LTI library to be used in a variety of contexts. We knew we could not have this library be context aware and know how to, say, retrieve a list of user roles in every place it's installed. Because the LTI implementation and the XBlock were coupled, the Django views were doing the job of retrieving all the contextual data from the platform and incorporating the data into the launch.

Our decision was to invert this pattern and to have users of the library supply a defined set of contextual data to the library before starting an LTI launch.

KEY COUPLING POINTS: CONTEXTUAL DATA

- Installing application supplies contextual data to Django plugin views.

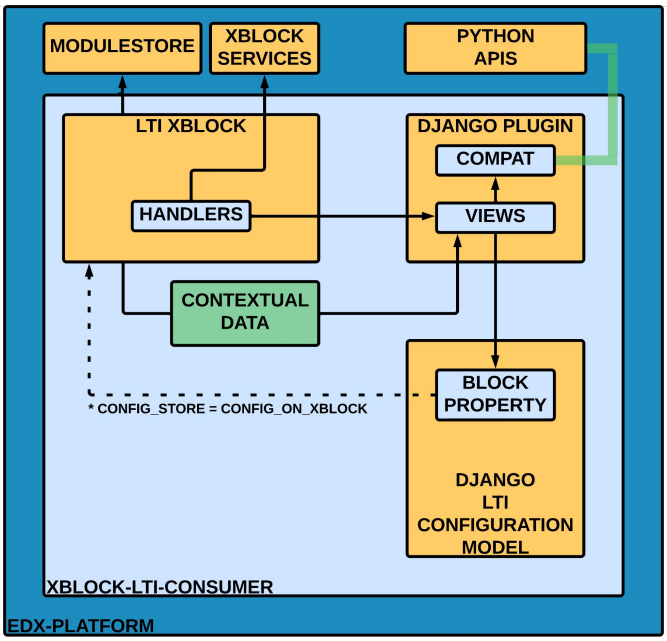


Now, instead of the plugin calling the platform APIs, it gets all the contextual data it needs from the installing context. In this case, that's the XBlock.

But what is this contextual data?

KEY COUPLING POINTS: CONTEXTUAL DATA

- Django plugin views no longer rely on platform Python APIs for contextual data.



By doing this, we can remove the dependency of the Django plugin on the platform Python APIs.

But what, exactly, is this contextual data?

CONTEXTUAL DATA

— We introduced a Lti1p3LaunchData data class to communicate contextual data.

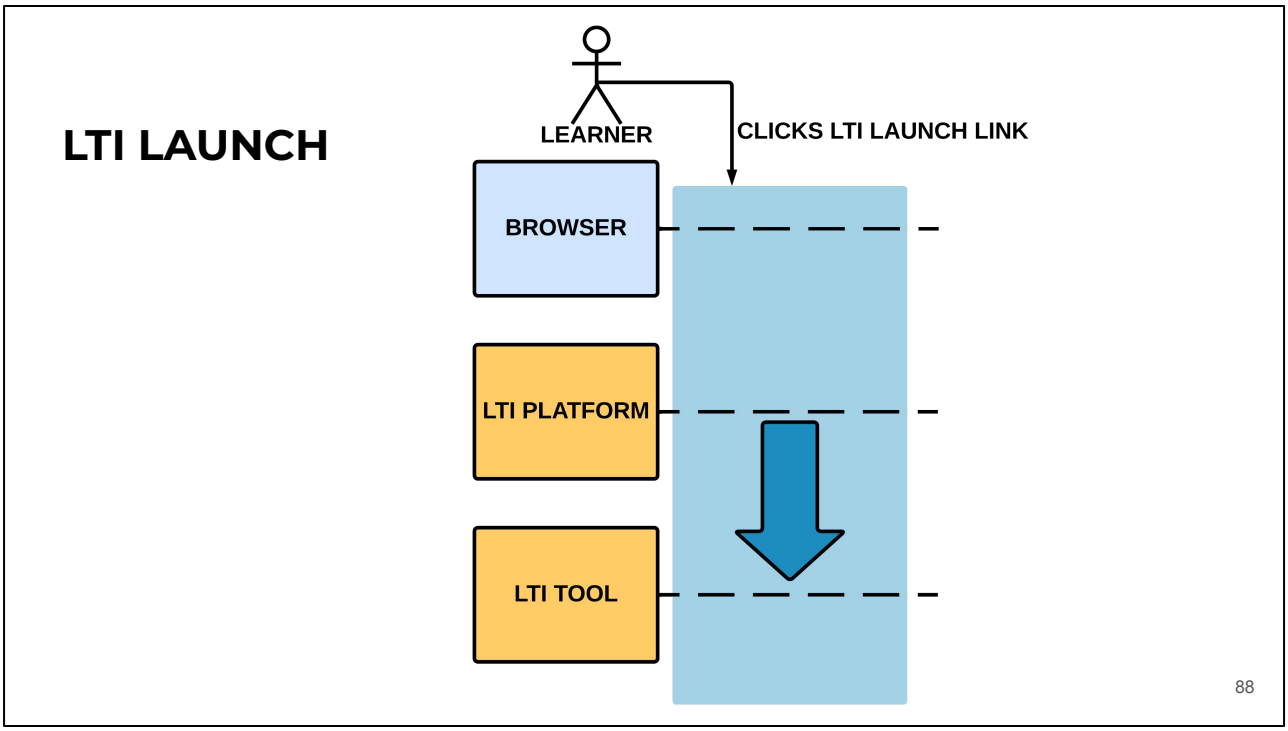
LTI1P3LAUNCHDATA

- **USER_ID**
- **USER_ROLE**
- **CONFIG_ID**
- **RESOURCE_LINK_ID**
- **PREFERRED_USERNAME**
- **EMAIL**
- **EXTERNAL_USER_ID**
- **MESSAGE_TYPE**
- **CONTEXT_ID**
- **CONTEXT_TYPE**
- **CONTEXT_TITLE**
- **CONTEXT_LABEL**

We created a simple data class to store all LTI contextual data that a user of the library would need to supply to the LTI library. The user of the library would import the data class, create an instance of it, and pass it to the LTI library to be available throughout the LTI launch flow.

But how does it actually get communicated from the installing application to the LTI library and then magically make its way through all of the LTI flow in a way that just “works”?

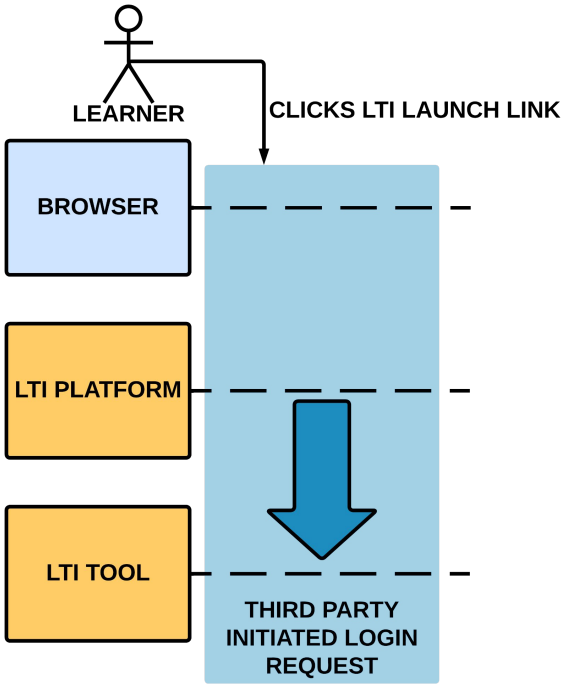
In order to understand this, we need to revisit how the LTI launch flow works.



Here is the LTI launch flow we discussed earlier. I've cut off the remainder of the launch flow to just focus on this initial part of the flow.

LTI LAUNCH

- This is the Third Party Initiated Login Request.



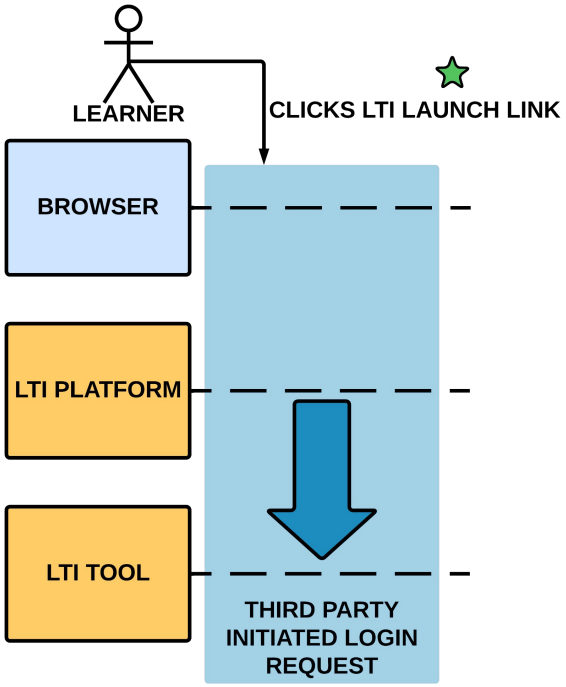
89

Recall that this is called the Third Party Initiated Login Request.

There's an important piece of the puzzle in this diagram, and I didn't explain it to you originally.

LTI LAUNCH

- LTI launch link is a link to the tool's Third Party Initiated Login URL.



90

I never actually explained to you what the "LTI Launch Link" is. What is this magical link that starts this entire flow?

Well, the third party initiated login request is what starts the LTI launch. Once that is done, the request and responses bounce between the Django plugin views and the tool through the browser. When the platform renders the page, it gets a URL to the tool's third party login endpoint that it needs to either make a GET or POST request to. This is the LTI launch link.

The structure of this URL is important.

LTI LAUNCH URL

www.tool.com/login?lti_message_hint=123

Here is a look at what a third party login endpoint URL could look like if we were going to make a GET request to it. The important pieces are the query parameters, which are described in the LTI standard. Of course, there are more parameters than just these, but the lti_message_hint parameter is the important to understanding the transmission of contextual data.

LTI LAUNCH URL

www.tool.com/login?**lti_message_hint**=123

92

The lti_message_hint is a parameter that is intended to supply context clues to the platform. After the third party initiated login request is made via the browser, the tool issues a response to the platform via the browser in the form of an authentication request. When the platform receives this authentication request, it has no idea what's going on. It has lost all of the context it had when it generated the original third party initiated login request. For example, it can no longer answer the questions, "Who was the user that requested an LTI launch?", "From where in the course content is this LTI launch occurring?", and so on. That's all lost.

The purpose of this parameter is for the platform to make whatever note of whatever it feels is appropriate to remind itself of this context. When the tool makes its authentication request to the platform, the standard requires that it include this two parameter unchanged.

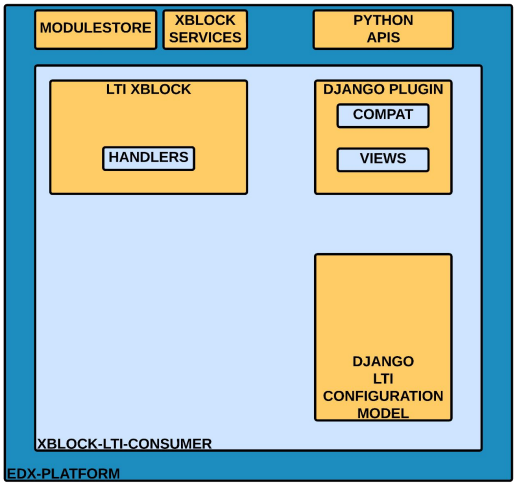
LTI LAUNCH URL

www.tool.com/login?lti_message_hint=**REF**

We decided that we'd use this query parameter to hold a reference to the data class that stores the contextual information. The user of the library would pass an instance of the contextual data data class to the API function that generates the login URL. The function would insert a reference to this data class into the login URL. When the LTI launch is started, this reference would be passed around through the flow. Whenever the LTI library would need access to the contextual data, it would use the reference to get access to the contextual data.

Let's take a look at what this looks like.

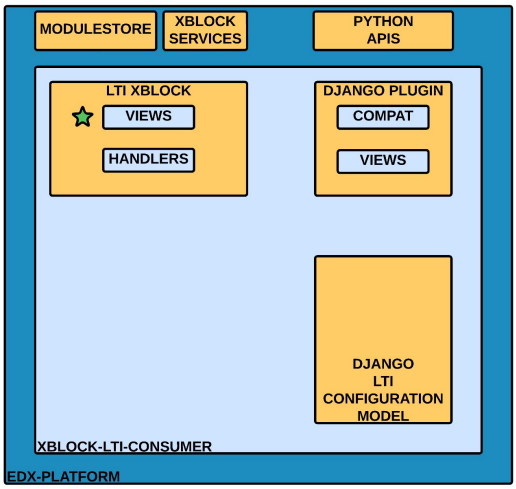
CONTEXTUAL DATA



Here we have the original key components of our LTI implementation. In order to understand how the LTI launch link is used, we need to fill in a few more components.

CONTEXTUAL DATA

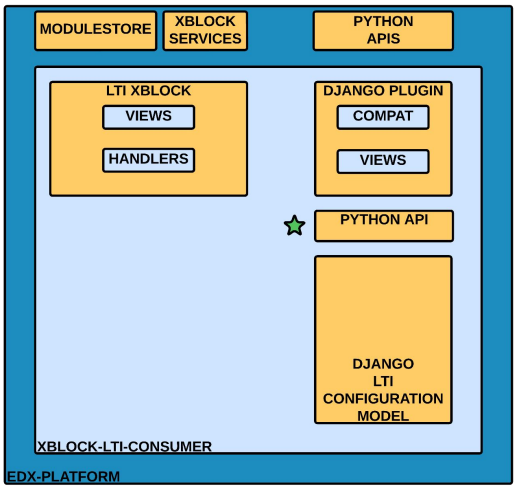
- XBlock has views for rendering the XBlock.
- student_view renders learner view in the LMS.



First, we have XBlock views. These are Python methods invoked by the XBlock runtime to render the XBlock. These are defined by the XBlock API. The *student_view* method renders what the learner sees in the LMS.

CONTEXTUAL DATA

- Python API function generates LTI launch URL.

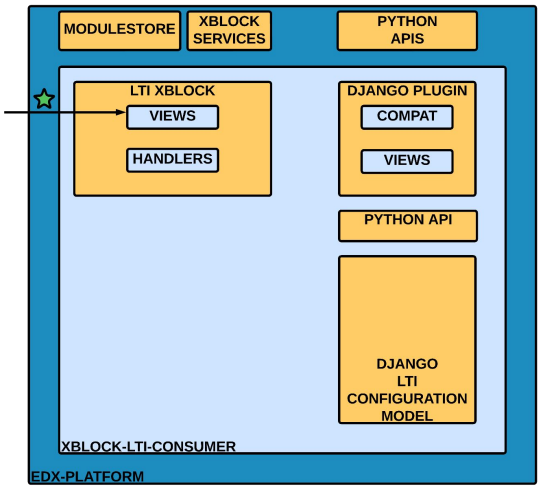


Next, we have a Python API in the lti-consumer-repository. It's contains the function that generates an LTI launch URL.

Now, let's take a look at what happens when a learner visits an LTI component in the LMS.

CONTEXTUAL DATA

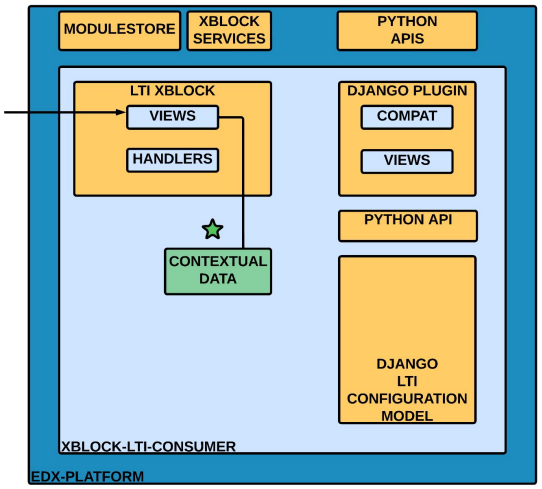
- XBlock student_view handles request to render the XBlock.



When a learner visits an LTI component, the *student_view* function is called to render the student view.

CONTEXTUAL DATA

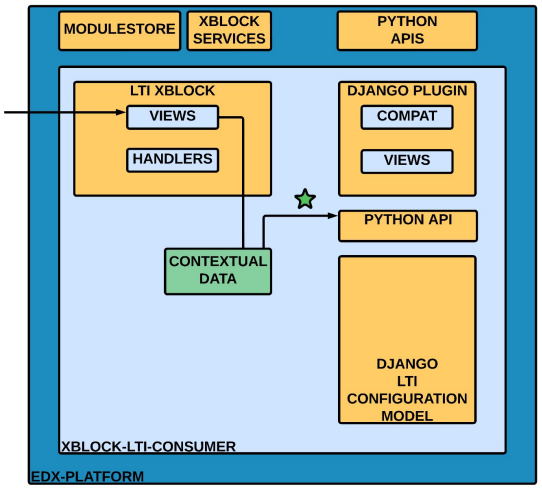
- student_view collects contextual launch data and instantiates Lti1p3LaunchData data class.



But what is this magical reference? The reference is a reference to the launch data. It had to somehow be stored by the LTI library before the third party initiated login URL so that it could be retrieved after receiving the authentication request. We went through a few iterations of what this reference could be.

CONTEXTUAL DATA

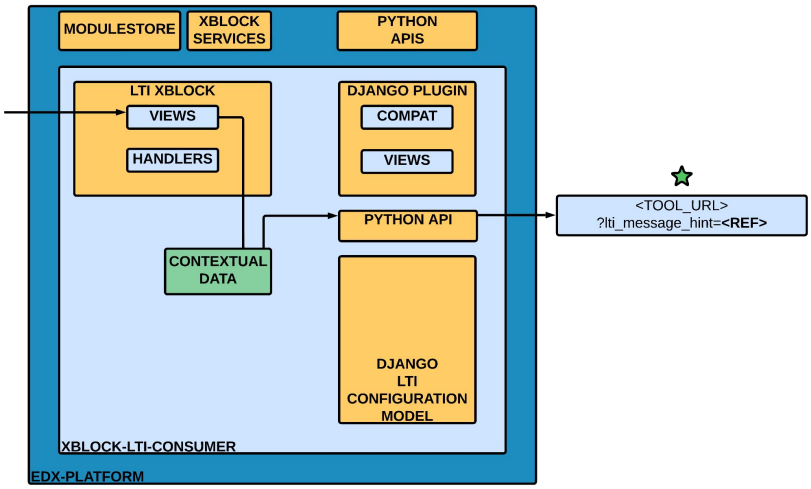
- student_view calls Python API function to generate LTI launch URL.



The *student_view* calls the Python API to generate the LTI launch URL.

CONTEXTUAL DATA

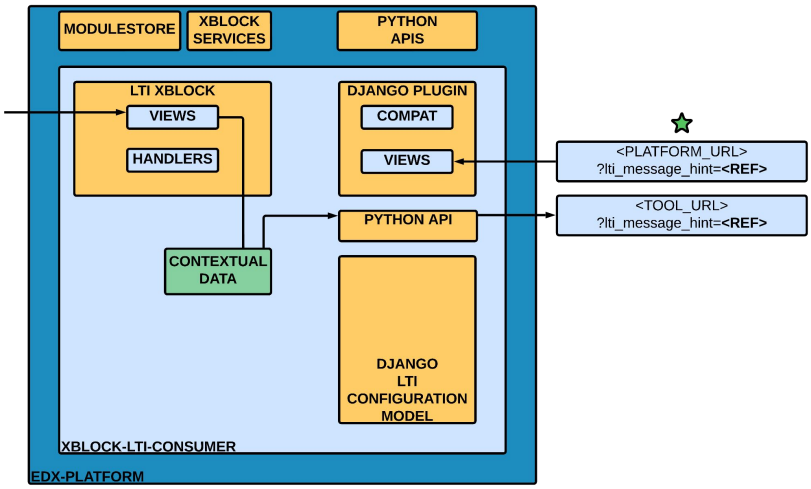
- LTI launch link is rendered in the courseware.
- Learner clicks LTI launch link, and the Third Party Initiated Login Request is made, sending the **lti_message_hint**.



The Python API generates an LTI launch URL, returns it to the *student_view*, and the launch link is rendered on the page. When the learner clicks the link, the LTI launch will begin. At this point, the platform has made the third party initiated login request via the browser.

CONTEXTUAL DATA

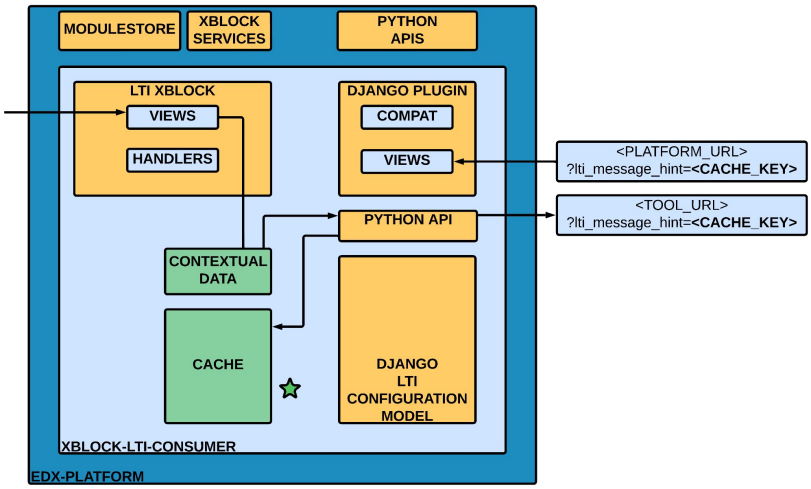
- Tool responds by making the Authentication Request, sending back the **lti_message_hint** unchanged.



Eventually, the tool will respond to the third party initiate login request with the authentication request via the browser. It makes a request to the platform's authentication endpoint. Notice that it sends back the *lti_message_hint* parameter unchanged

CONTEXTUAL DATA

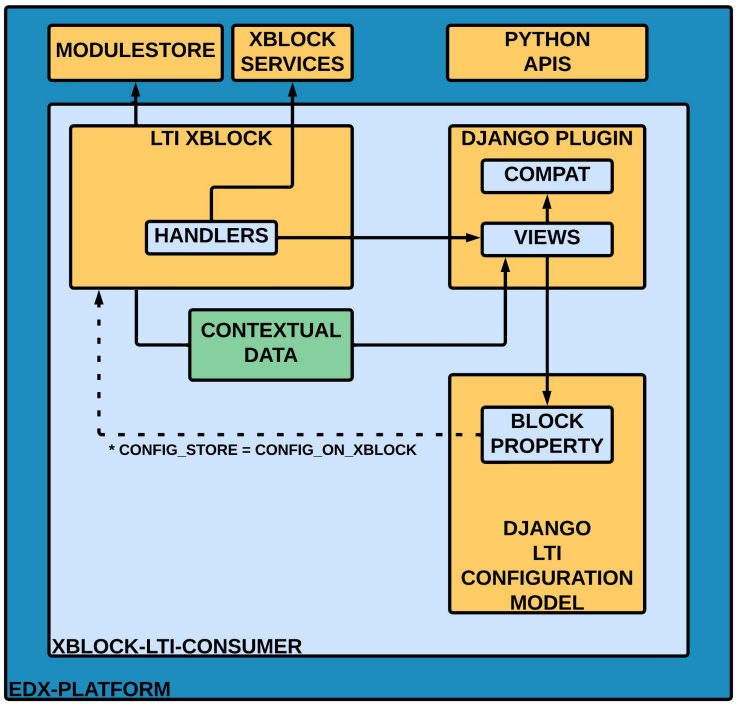
- Cache stores contextual data long enough for an LTI launch to occur.



We settled on storing the contextual data in a cache and including the cache key as the *lti_message_hint*.

FINAL STATE

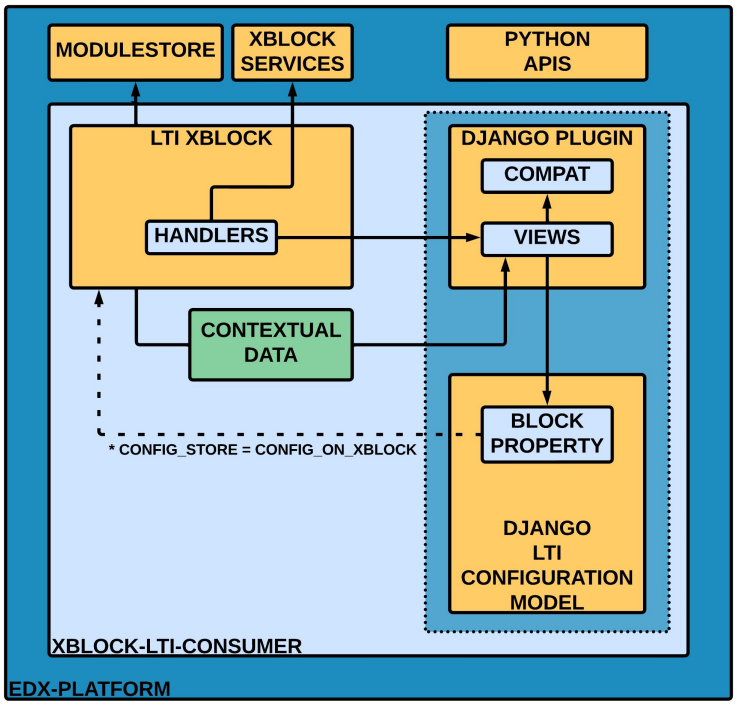
- This is the current state of the LTI library.



This is the final, current state of the xblock-lti-consumer library after all of the decoupling work.

FINAL STATE

- This is the current state of the LTI library.



Importantly, you'll notice that all the solid arrows are pointing from the user of the LTI library (i.e. the LTI XBlock) to the library, not vice-versa. This means that the LTI library is free to be used broadly across a variety of Djangoapps.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

You now understand what we did, but you may be wondering how you can leverage these changes to do an LTI launch from your application.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

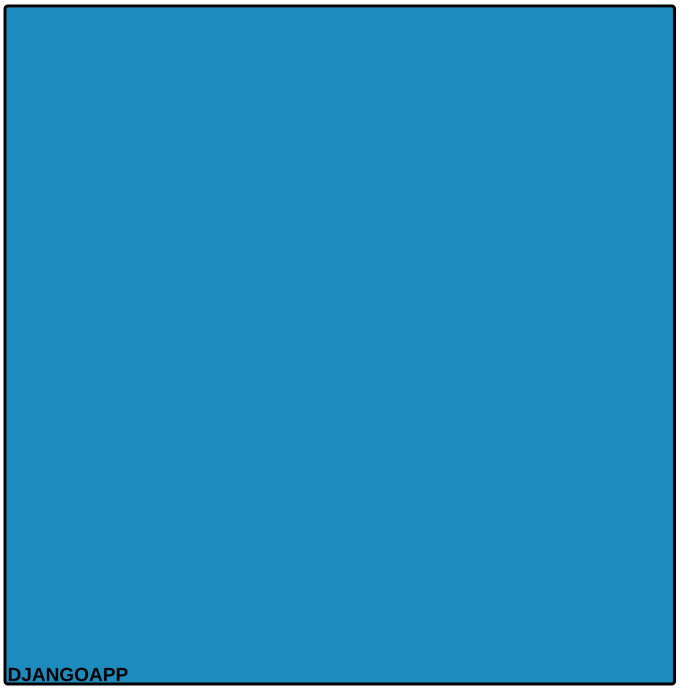
Next, we're going to learn how easy it is to set up an LTI launch in your Django application.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

There are six easy steps to setting up an LTI 1.3 launch in your Djangoapp.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

- This is your Djangoapp.
- This assumes you have a corresponding microfrontend.

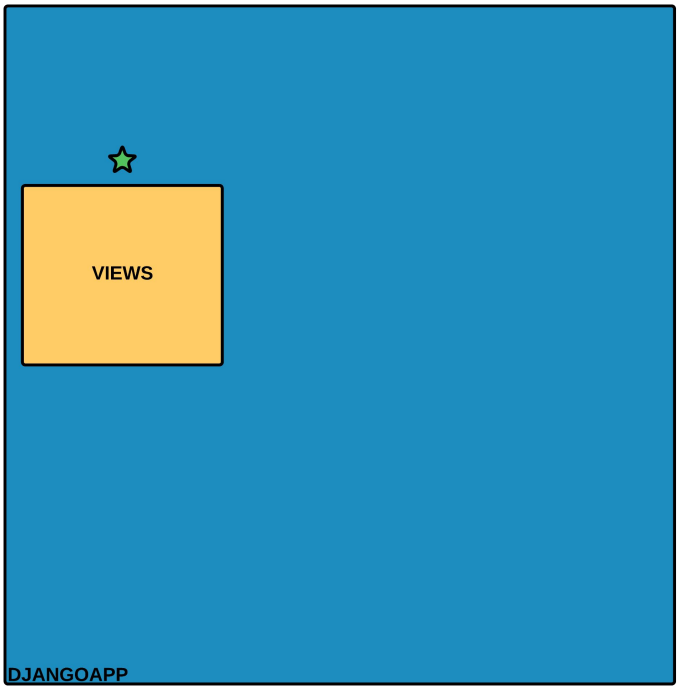


First, you have your Djangoapp.

This example assumes you have a corresponding microfrontend (MFE) and are not server-side rendering. It'll work just fine in the latter case; you just wouldn't send the LTI launch URL to the MFE in a backend-for-frontend (BFFE) endpoint response, for example.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

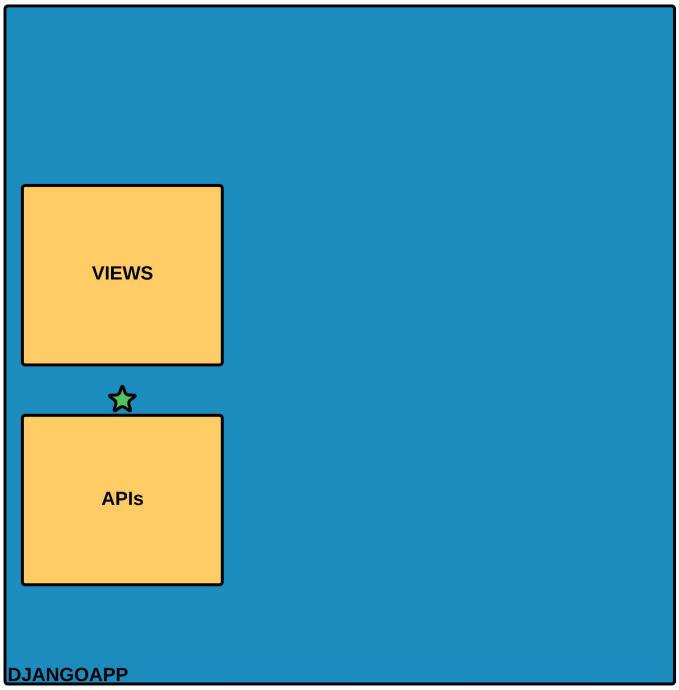
- Your Djangoapp will have views.
- Let's say it will have a BFFE endpoint that returns the LTI launch URL.



Your Djangoapp will have views. If you're running an MFE, this may be a backend-for-frontend (BFFE) endpoint that will return an LTI launch URL for the MFE to render to the learner.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

- Your Djangoapp will have contextual data to send via the LTI launch.
- Let's say it gets it from it's internal Python API.

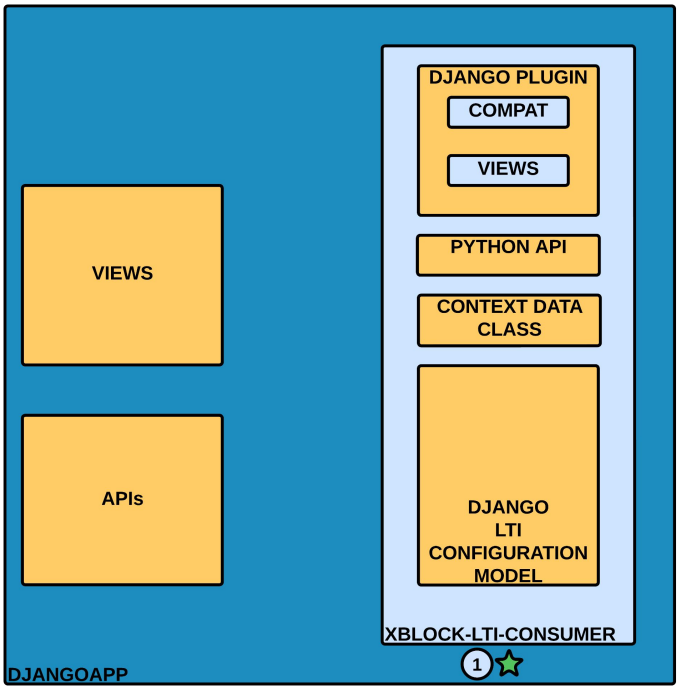


Next, you'll have contextual data about the LTI launch and some way to collect it. Let's say you do that using your Python API.

Now, we're ready to set up the LTI launch.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

1. Install the **xblock-lti-consumer** library into your Djangoapp.

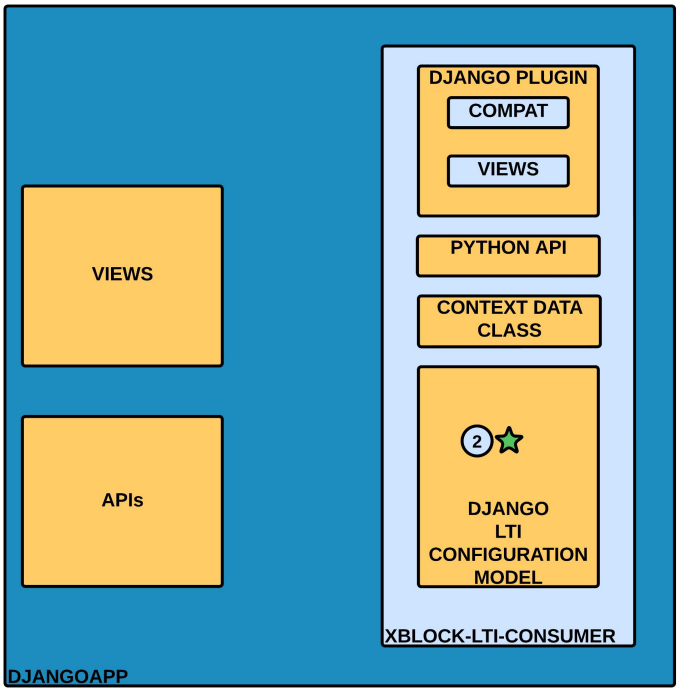


The first step is to install the xblock-lti-consumer library into your Djangoapp.

For simplicity's sake, I've only shown the relevant pieces here. But you will install the entirety of the library, including the LTI XBlock.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

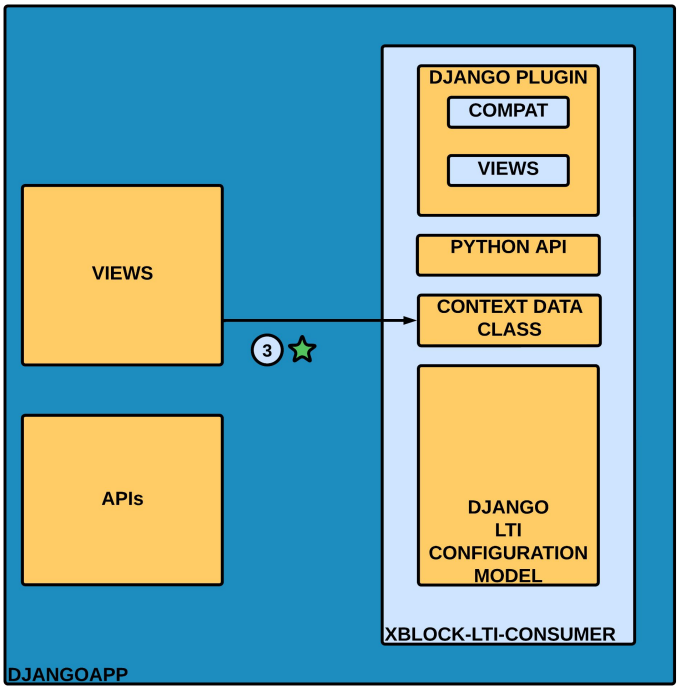
2. Store LTI configuration data in the **LtiConfiguration** Django model.



Next, you will store LTI configuration data in the xblock-lti-consumer's LtiConfiguration model. You will set the config_store field to CONFIG_ON_DB to indicate to the model that the full scope of LTI configuration data is stored on it. This way, it will not attempt to read from the modulestore or the block property.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

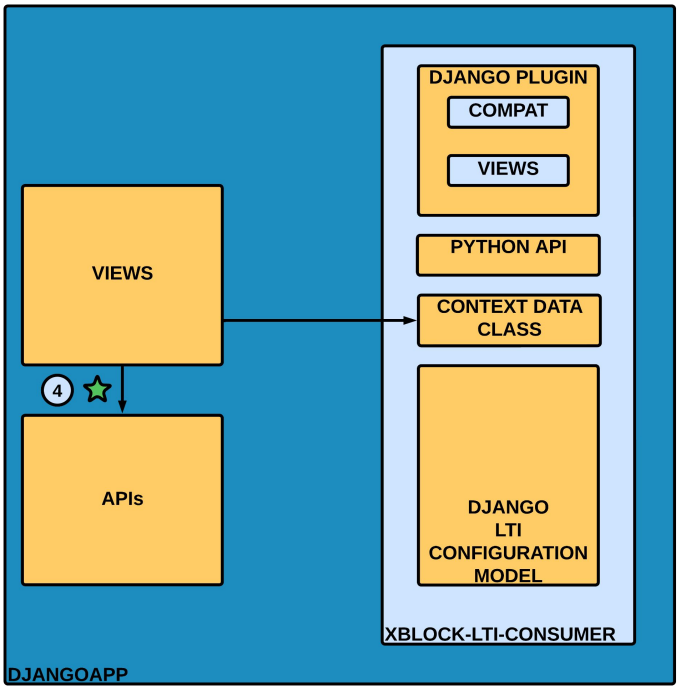
3. Import **Lti1p3LaunchData** data class from the data module.



Next, your view (or whatever code needs to generate an LTI launch URL) will import the Lti1p3LaunchData data class from the data module of the xblock-lti-consumer library.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

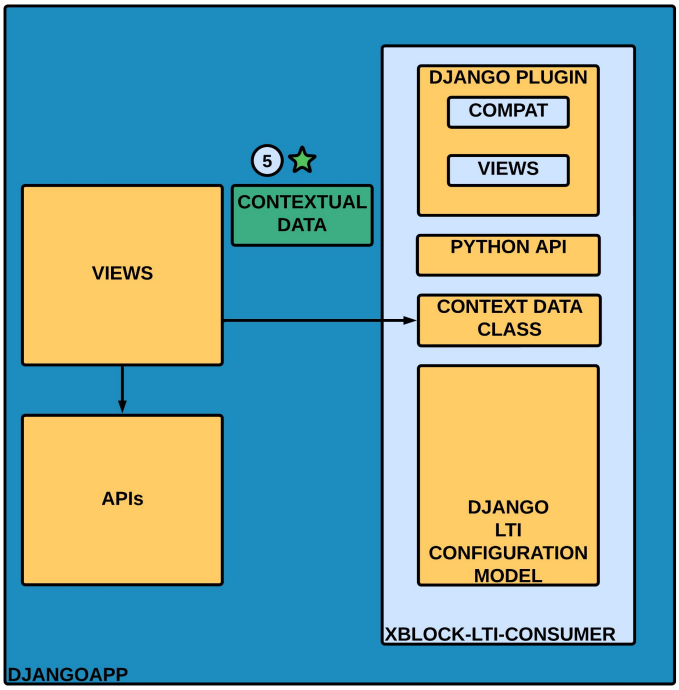
4. Collect contextual launch data.



Next, your view (or whatever code needs to generate an LTI launch URL) will call its Python APIs (or whatever code it uses to get contextual data) to collect the contextual data about the LTI launch.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

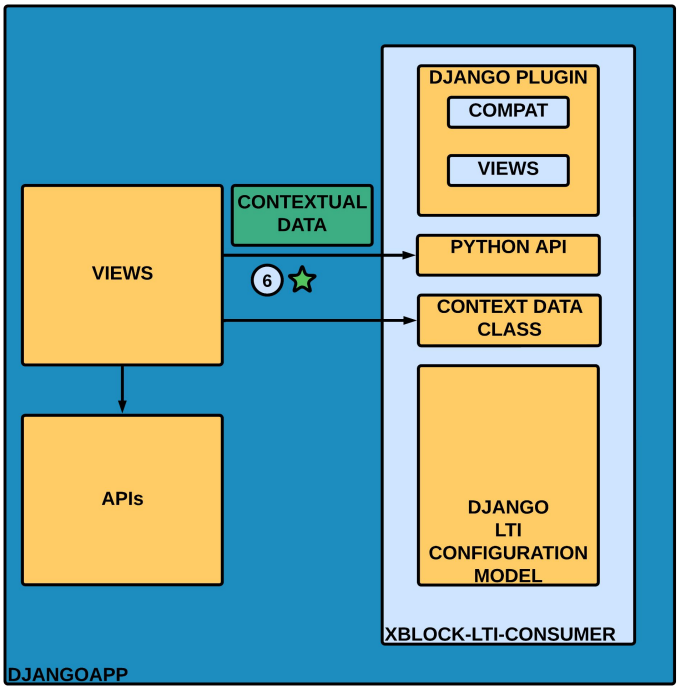
5. Create instance of **Lti1p3LaunchData** data class.



Next, your view (or whatever code needs to generate an LTI launch URL) will instantiate the Lti1p3LaunchData data class with the contextual data.

DJANGOAPP LTI 1.3 LAUNCH: 6 EASY STEPS

6. Call Python API function to generate LTI launch URL, passing in **Lti1p3LaunchData** data class instance.



Finally, your view (or whatever code needs to generate an LTI launch URL) will import and call the get_lti_1p3_launch_start_url function from the xblock-lti-consumer library's Python API, passing the instance of the Lti1p3LaunchData data class. This function will return an LTI launch URL. Your Djangoapp can return this to the MFE to be rendered to the learner. Or it can be redirected to from the Djangoapp.

Once the request associated with the URL is made, the LTI launch will begin, and the request handling will be done by the xblock-lti-consumer's Django plugin views.

DJANGOAPP LTI 1.3 LAUNCH: REVIEW

1. Install the LTI library.
2. Store configuration data in the **LtiConfiguration** model.
3. Import **Lti1p3LaunchData**.
4. Collect contextual data.
5. Instantiate **Lti1p3LaunchData** with contextual data.
6. Pass the instance to **get_lti_1p3_launch_start_url** API function to get LTI launch URL.

As a review, these are the six easy steps to set up an LTI 1.3 launch in your Djangoapp.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

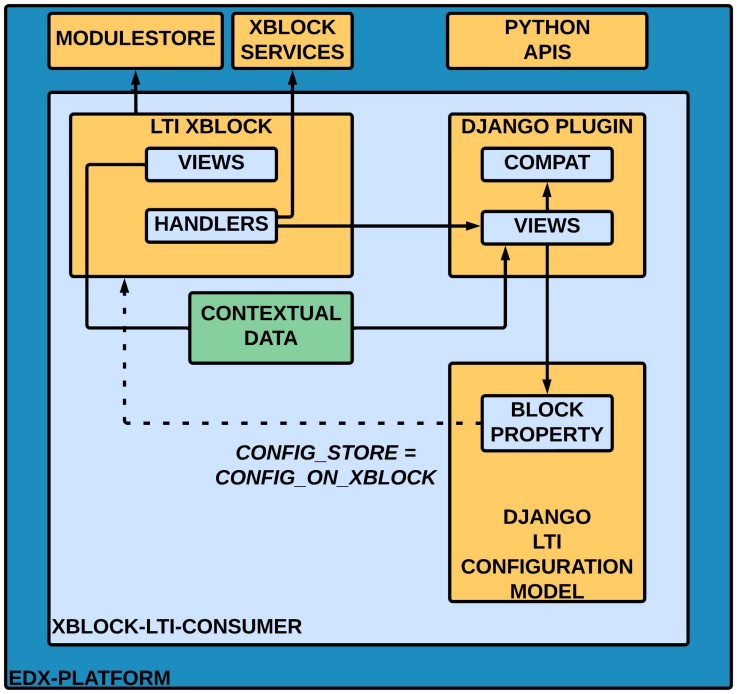
We've reviewed the six easy steps to set up an LTI 1.3 launch in your Djangoapp.

AGENDA

1. What is LTI?
2. What was the problem?
3. What was our approach?
4. What's the impact?
5. How did we do it?
6. How can you do an LTI 1.3 launch from your application?
7. How can you help?

Now, let's end with how you can help or get involved. This will be a discussions of the next steps we need to take to further improve our LTI implementation.

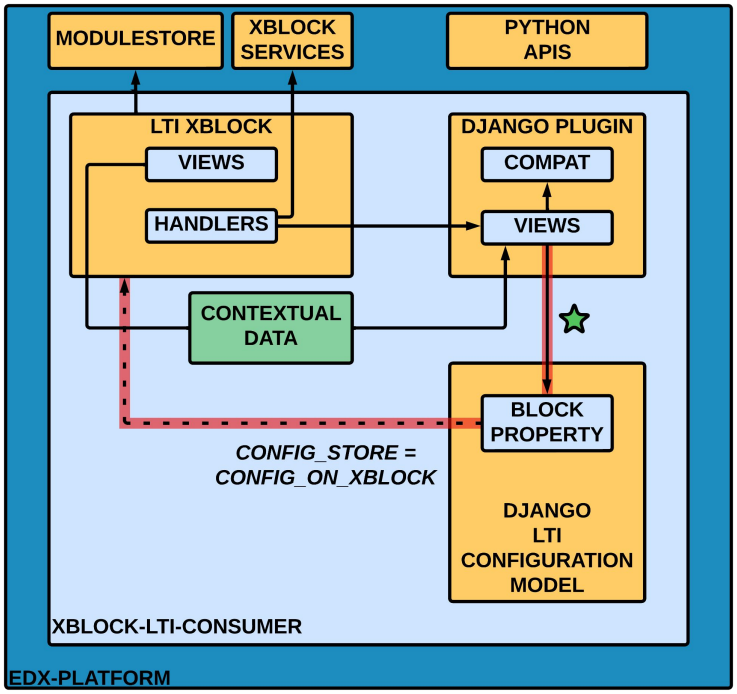
NEXT STEPS



This is a review of the current state of our LTI implementation.

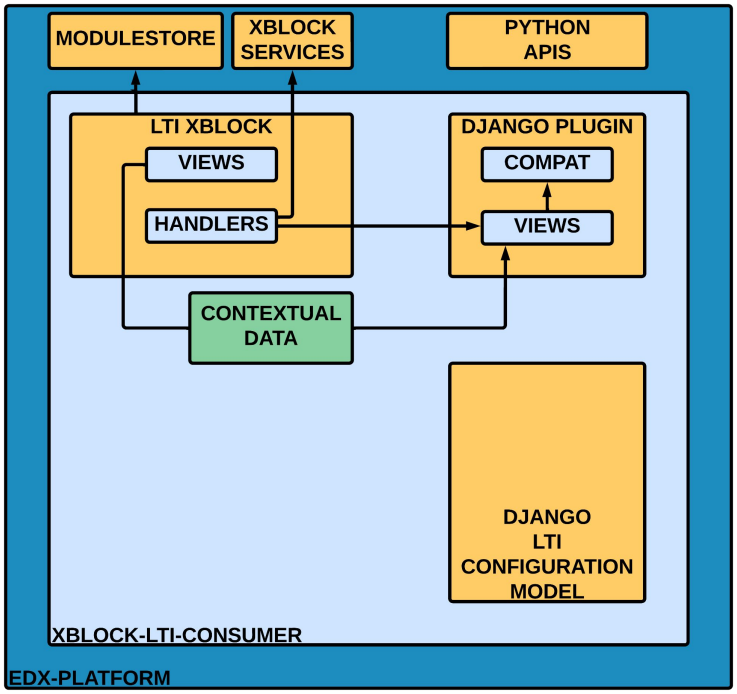
NEXT STEPS

- Remove LTI configuration data from the XBlock and the modulestore.
- Store LTI configuration data in the Django model.



We would like to remove LTI configuration data from the modulestore. Instead, we'd like to store LTI configuration data in the Django LTI configuration model (or in a plugin via the filter hook, as described by the CONFIG_EXTERNAL config_store option).

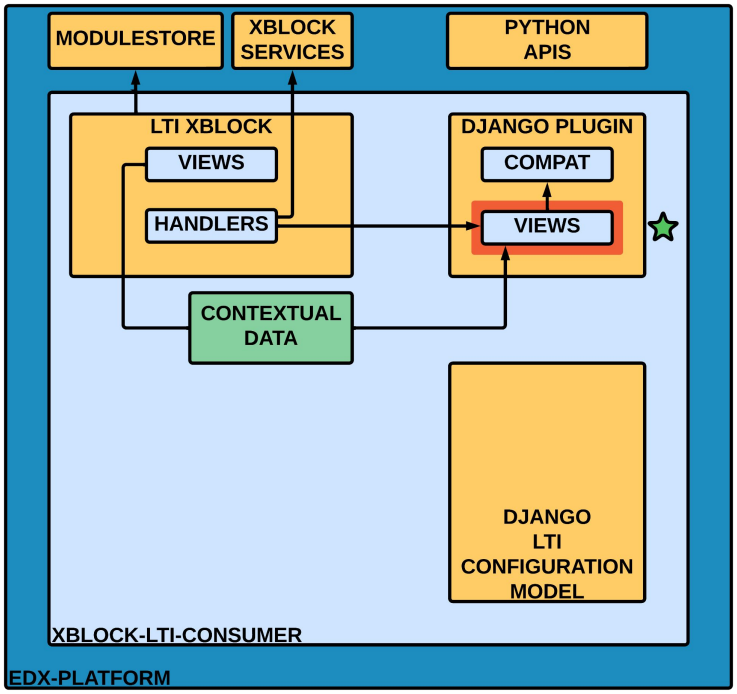
NEXT STEPS



If we do this, we can remove this dependency entirely. The Django LTI configuration model no longer needs a block property.

NEXT STEPS

- Decouple LTI Advantage Services views in the Django plugin views.

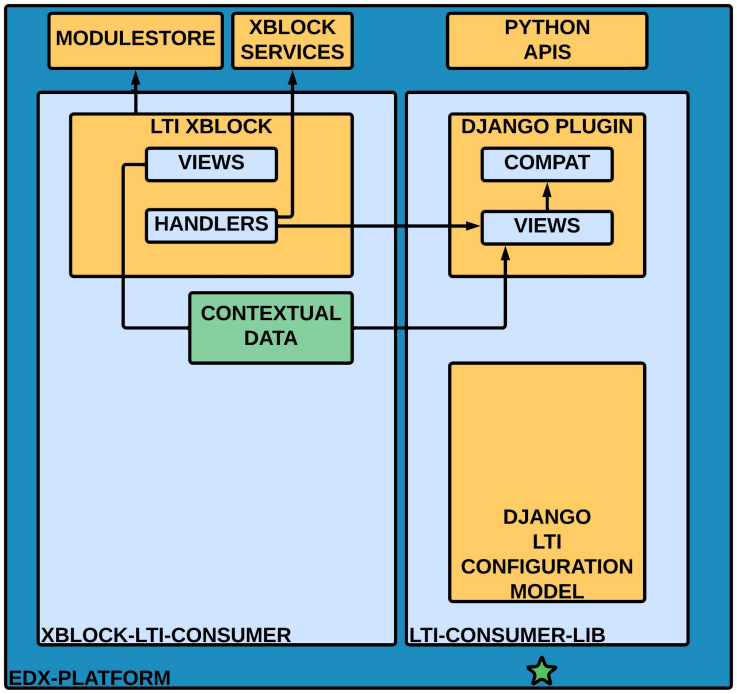


123

I mentioned we would not be discussing LTI Advantage Services today. That's because the views that implement these service endpoints suffer from the same kind of coupling. We'd like to fully decouple these service endpoints as well.

NEXT STEPS

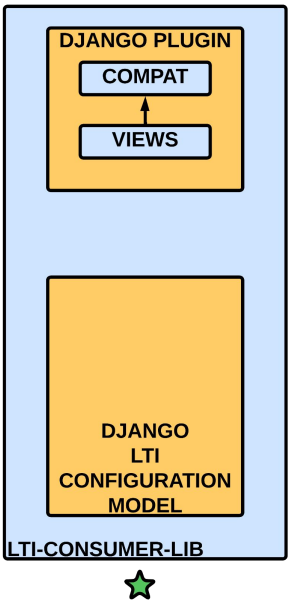
- Separate LTI implementation into LTI XBlock and LTI library.



Once we've done this, we can create a firmer boundary between the LTI XBlock and the LTI library.

NEXT STEPS

- Provide stand-alone LTI library for use across the edX platform.



And, eventually, we can break the LTI library out into its own installable library, providing us an independent, core implementation of LTI that can be used throughout the ecosystem.

GETTING IN TOUCH

- **EMAIL**
  - mroytman@2u.com
- **OPEN EDX SLACK**
  - @mroytman
  - #lti
  - #lti-1-3

That brings us to the end of today's talk.

If you have any further questions, comments, or concerns, please feel to reach out at any time.

QUESTIONS?