
FRONT-END DEVELOPER TRAINING

for

LIFERAY DXP



Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

7.0.5.1

Contents

1 Development Environment	7
1.1 Course Schedule	7
1.2 Front-End Development in Liferay	10
1.3 Front-End Development Tools	13
1.4 Setting Up the Development Environment	19
1.5 Front End Development for S.P.A.C.E.	35
2 New UI Landscape	39
2.1 Liferay's UI Technologies	39
2.2 Introducing Lexicon	44
2.3 JavaScript in Liferay	48
2.4 Custom Displays with Templates	52
3 Theme Development	57
3.1 Styling Liferay with Themes	57
3.2 Using the Liferay Theme Generator	67
3.3 Controlling Page Structure	82
3.4 Styling the Platform	94
3.5 Adding Custom JavaScript	109
3.6 Configuring the Theme	117
3.7 Themelets	130
3.8 Theme Contributors	139
3.9 Importing Resources in a Theme	145
3.10 Embedding Applications into Themes	157
4 Layout Templates	165
4.1 Controlling Page Layouts	165
4.2 Generating Layout Templates	171
4.3 Typical Elements and Classes	179
4.4 Embedding Applications in Layout Templates	183
5 Customizing the Front-end	187
5.1 Application Display Templates in Liferay	187
5.2 Template Language Options	192

5.3	Using Lexicon in Templates	199
6	Delivering Consistent Content Experiences	205
6.1	Styling Content with Web Content Templates	205
6.2	Creating Content Templates	211
6.3	Using Preferences in Templates	226
7	Customizing Workflow Email Notifications	245
7.1	Customizing Notification Templates	245
7.2	Getting Information from Workflow	249
8	Creating Different Front Ends for Applications	259
8.1	Customizing Application Displays	259
8.2	Styling Applications	264
Appendix - Lexicon Details		275
8.3	Lexicon Details	275
Appendix - Modern Application Design with Templates		283
8.4	Metal.js Soy	283
8.5	Metal.js JSX	291
8.6	Basic Soy Syntax	296
8.7	Building Application UIs with Soy	307
Appendix - Customizing the Alloy Editor		317
8.8	Alloy Editor in Liferay	317
8.9	Configuring And Customizing Allowed Content	322
8.10	Customizing the Toolbar Options	326
8.11	Using the Alloy Editor in Applications	331
8.12	Adding Buttons to the Editor	334
Appendix - JavaScript		339
8.13	Liferay AMD Module Loader	339
8.14	JSON Web Services	346
8.15	Liferay Internal APIs	354
8.16	JavaScript - Single Page Application	362
8.17	Loading Scripts with the AUI Script Tag	370
8.18	Theme Display Object	373
8.19	Modern Web Experiences: ECMAScript 2015	376
8.20	Building Components: Metal.js	381
8.21	Alloy UI in Liferay 7	385
Appendix - Portlets		389
8.22	Portlets - Introduction	389
8.23	Application Display Template Usage In Custom Portlets	393
8.24	Localization	397
8.25	Styling the Application Container	403

Appendix - Taglibs	407
8.26 Taglibs - Introduction	407
8.27 Forms and Validation	411
8.28 Taglibs - Page Layout	418
8.29 Taglibs - UI Components	421
8.30 Taglibs - Utility Components	424
Appendix - Upgrading a Theme to DXP	433
8.31 Upgrading a theme to dxp	433

Chapter 1

Development Environment



COURSE SCHEDULE

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

DAY 1

- Development Environment
 - Using Front-End Development Tools
- New UI Landscape
 - Lexicon CSS
 - ECMAScript 2015 Features
 - Building Components: Metal.js
 - Customizing Alloy UI
- Theme Development
 - Creating a Theme
 - Using Themelets
 - Packaging Content in Themes

DAY 2

- ❖ Layout Templates
- ❖ Customizing the Front-End with Templates
- ❖ Delivering Consistent Content Experiences
 - Styling Web Content with Templates
- ❖ Branding Email Notifications
 - Creating Advanced, Styled Workflow Notifications
- ❖ Creating Different Application Front-Ends

CLASS SCHEDULE

- ❖ The class will be divided up into sessions of approximately 90 minutes or less.
- ❖ There will be shorter breaks at appropriate stopping points throughout the day, and a longer break for lunch.
- ❖ We'll have 10 minutes for Q&A after each section. This is a good time to ask questions about recently-covered topics or how covered topics relate to each other.
- ❖ We'll also have 30 – 60 minutes for Q&A at the end of each day. This is a good time to ask questions that are not covered in the course topics.



FRONT-END DEVELOPMENT IN LIFERAY

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

INTRODUCTION TO FRONT-END DEVELOPMENT

- Liferay DXP is a fully customizable platform.
- Front and back-end developers have a wide variety of options for both functional and stylistic customizations.
- In this course, we'll look at styling options with theme modules and templates.
- We'll be talking about both general and specific look-and-feel changes to match your business requirements.

STYLING OPTIONS

- ❖ Liferay has two general options for stylistic changes:
 1. **Modules:** include Themes and Layout Templates for global look-and-feel changes
 2. **Templates:** include specific customization options for content, applications, and email notifications

Script

```
Search _____
```

General Variables

Device
Portal Instance
Portal Instance ID
Site ID
View Mode

```
1 <style>
2
3 <img>.display {
4   width: 200px;
5   height: 150px;
6 }
7
8 <div>_Frontpage {
9   background-image: url(${Image6kp8.getData()});
10  color: #333333;
11  padding-top: 50px;
12  padding-bottom: 50px;
13 }
14
15 <h2>-title {
16   font-size: xx-large;
17   font-variant: small-caps;
```

WWW.LIFERAY.COM

LIFERAY.

DEVELOPING WITH STYLE

- ❖ As a platform, Liferay contains a number of frameworks and tools that make rapid development possible.
- ❖ Let's take a look at some of the tools we can use to beautify the platform.

The screenshot shows a Liferay website for "Space Program Academy of Continuing Education". The header includes navigation links like "Home", "About", "Programs", "Events", "Contact", and "Log In". Below the header, there's a banner with the text "Dare to dream out of this world." and an image of a modern building with a curved glass facade. The main content area features a section titled "News" with articles about space research and student profiles. There are also three smaller modules: "Dream even bigger", "Find your future", and "Prepare for lift-off".

WWW.LIFERAY.COM

LIFERAY.

Notes:



FRONT-END DEVELOPMENT TOOLS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

DEVELOPING IN LIFERAY

- ❖ Liferay is tool-agnostic.
- ❖ Anything from a command prompt or text editor to a full-blown IDE can be used to develop on Liferay.
- ❖ Liferay DXP introduces a number of new tools we can take advantage of to simplify development.
- ❖ Customizations and new features in Liferay are deployed in *modules*.
- ❖ Themes and Layout Templates are installed as modules.
- ❖ Previously, front-end developers needed to use the Plugins SDK.
- ❖ If you'd like to know how to set up Liferay with another development environment, our online Developer Guide covers this process:
https://dev.liferay.com/develop/reference/-/knowledge_base/7-0/development-reference

RIGHT TOOL FOR THE JOB

- ❖ Some tools will simplify creating, building, and testing *modules* in Liferay.
- ❖ There are tons of helpful tools for front-end development – you might have a few favorites already.
- ❖ We'll use a few standard tools for creating our projects:
 - *Node.js*
 - *gulp*
 - *Yeoman*
- ❖ You may already be familiar with these or similar tools.
- ❖ If not, don't worry: they're easy to learn and use.

TOOLS FOR CREATING

- ❖ We need a base *platform* for building and testing themes.
 - **Node.js** provides the platform.
- ❖ We want to easily *create* new projects.
 - **Yeoman** gives us that power.
- ❖ We want to *build* and *deploy* our projects to test them.
 - **gulp** makes those tasks a breeze.

NODE.JS AND NPM

- ❖ **Node.js** is a JavaScript runtime environment and will provide the platform we need to create and test our themes.
- ❖ With *Node.js*, we have access to a vast library of JavaScript packages.
- ❖ These libraries are accessed via *NPM*, the *Node.js* package manager.
- ❖ *NPM* helps users install and manage the packages and dependencies needed for the projects they are working on.
- ❖ This automation lends itself to a quicker, more efficient workflow.
- ❖ With *Node.js* and *NPM*, we'll have access to the tools needed to get our projects started.
 - If you'd like a refresher on *Node.js*, or want more details, check out <http://nodejs.org>.

CREATING WITH YEOMAN

- ❖ **Yeoman** is a powerful tool that runs on *Node.js*.
- ❖ Liferay modules are complex to create from scratch.
- ❖ We don't need to know the technical details; we just want to start building.
- ❖ *Yeoman* lets us create from templates using *generators*.
- ❖ A *generator* is a recipe for creating a new project from scratch – just add water!
- ❖ We'll look at installing a custom Liferay generator in Yeoman later.
- ❖ *Yeoman* makes sure projects are created the right way every time.

BUILDING WITH GULP

- Once we've created a project, we need to know how to:
 - Build
 - Package
 - Deploy
- **gulp** runs tasks with a simple command like:
`gulp build`
- All the details are handled behind the scenes.
 - You may already be familiar with other *Node.js*-based build tools like *Grunt*.
- *gulp* does all the hard work so we can *build* and *deploy* themes easily.

DEVELOPMENT TOOLS FOR MODULES

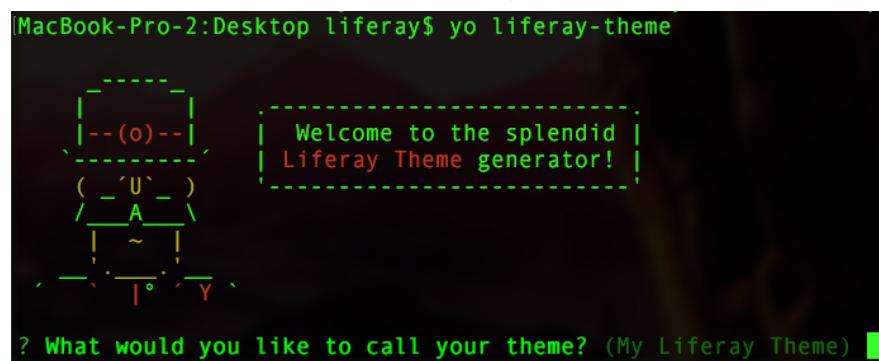
- So here's everything we need to style Liferay:
 1. **Java Development Kit (JDK) 8:** used to run Liferay and our installer. We'll also set the `JAVA_HOME` environment variable.
 2. **Node.js (packaged with npm):** used to install our Liferay Theme Generator and dependencies
 3. **Yeoman and gulp:** global dependencies, used to run the Liferay Theme Generator and `gulp liferay-theme` tasks
 4. **Liferay Theme Generator:** used to generate our themes and themelets
 5. **Brackets:** Our text editor of choice used to edit our theme files
 6. **Liferay Tomcat Bundle:** used to view development changes on our local machines

THEME BUILDER GRADLE PLUGIN

- ❖ We'll be using *Liferay Theme Generator* and the Node.js platform in exercises throughout the course.
- ❖ As an alternative Liferay also offers the Theme Builder Gradle plugin for theme development.
- ❖ For more information on using the Theme Builder Gradle plugin you can reference documentation at https://dev.liferay.com/de/develop/reference/-/knowledge_base/7-0/theme-builder-gradle-plugin

USING THE TOOLS WE'VE BEEN GIVEN

- ❖ Templates can be developed on the platform, as we'll explore later.
- ❖ Next, we'll set up our tools for developing Themes and Layouts.



```
MacBook-Pro-2:Desktop liferay$ yo liferay-theme
      _.-~.
     (   'U' )
      \_A\_
        | ~ |
       / - \_
         \_o_/
           Y
[Welcome to the splendid
 Liferay Theme generator!]
? What would you like to call your theme? (My Liferay Theme)
```

Notes:



SETTING UP THE DEVELOPMENT ENVIRONMENT

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

INSTALLING JAVA

- ❖ You should already have the Java JDK 8 installed.
- ❖ If not, you can download the latest JDK from Oracle:
<http://www.oracle.com/technetwork/java/javase/downloads/index.html>
- ❖ The Java JDK 8 is necessary to run and develop on Liferay.
- ❖ If you need to install JDK 8, please complete installation before continuing set-up.

EXERCISE: SETTING ENVIRONMENT VARIABLES ON WINDOWS

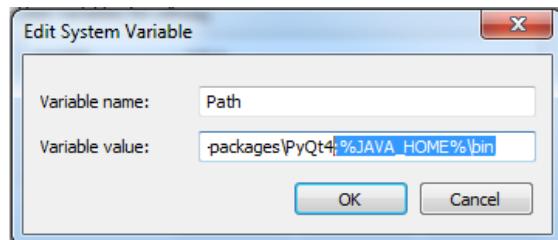
1. **Go to** Windows→Control Panel→System.
2. **Click** on the Advanced System Settings link in the left panel.
3. **Click** on the Advanced tab in the System Properties dialog box.
4. **Click** the Environment Variables button.

EXERCISE: FINISHING WINDOWS SET-UP

1. **Click** New under System Variables.
2. **Type** JAVA_HOME in the variable name field.
3. **Type** the install path for the JDK in the variable value field. (e.g., C:\Program Files\Java\jdk8)
4. **Click** OK.

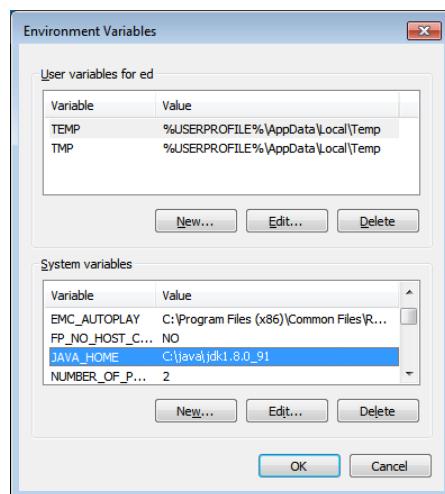
EXERCISE: ADDING TO THE PATH

1. **Find** the variable *Path* under *System Variables*.
2. **Click** *Edit*.
3. **Add** ;%JAVA_HOME%\bin to the end of *Path*.
4. **Click** *OK*→*OK*→*OK*.



EXERCISE: VERIFYING JAVA_HOME

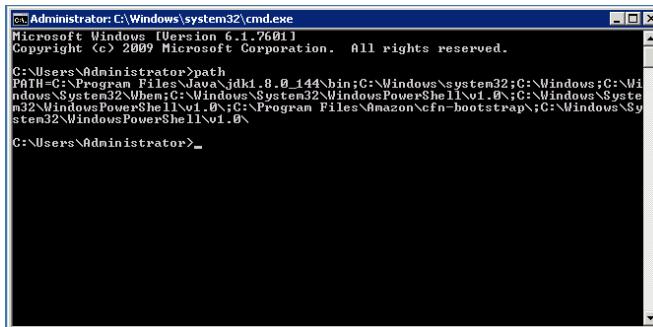
- Make sure that *JAVA_HOME* is correct.



EXERCISE: VERIFYING THE PATH ON WINDOWS

1. **Click** Start→Run...
 2. **Type** cmd.
 3. **Press** Enter.
 4. **Type** path.

✓ There should be only one JDK in the path.



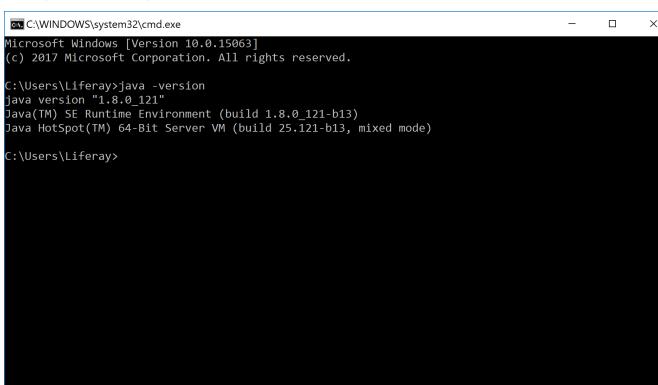
WWW.LIFERAY.COM

LIFERAY

EXERCISE: VERIFYING THE JAVA VERSION

1. Click Start→Run...
 2. Type cmd.
 3. Type java -version.

✓ The following message should look similar to this:



WWW.LIFERAY.COM

LIFERAY

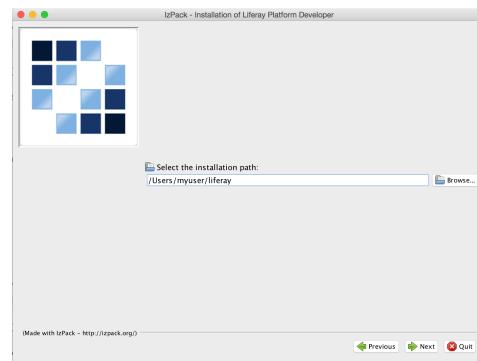
EXERCISE: INSTALLING EXERCISE FILES

- In your exercise materials, you'll find an installer labeled `lrurel-front-end-developer-installer-[version].jar`.
- This installer will install all the hands-on exercise files for the course.

1. **Run** the installer.
2. **Double-click** the file to run it if you're running Windows.
3. **Go to** the file in the terminal and run the installer using this command if you're on Linux or Mac:
`java -jar lrurel-front-end-developer-installer-[version].jar`

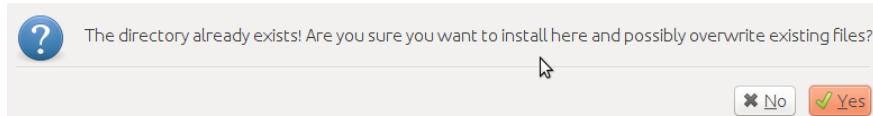
EXERCISE: INSTALLATION PATHS

1. **Click** through the steps to install the exercise files and accept the terms of use.
2. **Click** *I accept the terms of this license agreement.*
3. **Click** *Next.*
4. **Choose** the folder path where we will install our Liferay bundle.
 - **Mac:** should be `/Users/[your-user]/liferay`
 - **GNU/Linux:** should be `/home/[your-user]/liferay`
 - **Windows:** should be `C:\liferay`
5. **Click** *Next.*



EXERCISE: WARNING!

- A warning will pop up, cautioning us that we may overwrite existing files.
- 1. **Click Yes.**
- Don't worry; none of the things that we've done so far will be affected.



- 1. **Click Next→Next→Done.**
- ✓ Now we have our exercise files!

EXERCISE: INSTALLING NODE.JS

1. **Find** the node-[os]-[version] installer located in *exercises/front-end-developer-exercises/01-development-environment*.
2. **Run** the installer for your operating system and follow the instructions.
3. **Open** a *Command Prompt/Terminal* window.
4. **Run** `node -v` in your command line to verify your installation once you have installed `node.js` and `npm`.
- ✓ You should see something similar to the following output:
`v6.x.x`
5. **Run** `npm -version`.
- ✓ You should see the following output:
`3.X.X`

TROUBLESHOOTING: ADDING NODE.JS TO THE PATH

- ❖ The Node.js installer should add the install path to the PATH environment variable, but some Windows users may still have to manually set the variable.
- ❖ The PATH variable can be set in the *Advanced System Settings* menu.
 1. **Open** Window's Control Panel.
 2. **Go to** System→Advanced System Settings.
 3. **Click** on Enviroment Variables.
 4. **Double-click** on the Path system variable to edit.
 5. **Click** New.
 6. **Type** in the install path for Node.js.
 - This is typically installed to C:/Program Files/nodejs/
 7. **Click** OK.

EXERCISE: INSTALLING DEPENDENCIES

- ❖ Now that we have our npm settings configured, we can install the dependencies for the Liferay Theme Generator.
 - ❖ **Note:** When using the *global* option (-g) with npm, you need to have Administrator access on Windows, and you need to use sudo on UNIX-type systems.
1. **Open** the command line.
 2. **Type** the following command to install the Yeoman and gulp global dependencies:
`npm install -g yo gulp`
- ✓ With our dependencies installed, we can move on to our Liferay Theme Generator next.

EXERCISE: INSTALLING THE LIFERAY THEME GENERATOR

1. **Run** the following in the command line to install the generator:
`npm install -g generator-liferay-theme`
2. **Type** `yo` in the command line to see that the generator is installed.
3. **Press Enter.**
4. **Type** `Y` or `N` when the initial prompt asks to collect data.
 - This should only appear the first time you run Yeoman.
5. **Press Enter.**
 - You should see that *Liferay Theme* is listed underneath the heading *Run a generator*.
6. **Choose** `Get me out of here!` to close the prompt.

TROUBLESHOOTING: A LOCAL INSTALLATION

- If you don't have admin access, you need to create a directory and install everything locally.
 - **Note:** You need full Administrator privileges over this directory.
1. **Open** the command line.
 2. **Go to** the liferay directory created by our installer.
 3. **Type** the following command to install the Yeoman and gulp dependencies:
`npm install yo gulp`
 4. **Run** the following command to install the generator:
`npm install generator-liferay-theme`
- ✓ Now you'll be able to use the Liferay Theme Generator in this local directory.

TROUBLESHOOTING: NPM PERMISSIONS

- ❖ In some cases, there may be permission issues when using npm packages.
- ❖ There are three options for addressing possible issues:
 1. Change the permission to npm's default directory
 2. Change npm's default directory to another directory
 3. Use a package manager that takes care of the permissions for you
- ❖ You can find more details here:
<https://docs.npmjs.com/getting-started/fixing-npm-permissions>

LIFERAY-TOMCAT BUNDLE

- ❖ The Liferay-Tomcat bundle contains everything that is needed to run Liferay. No need to worry about additional set-up or configuration.
- ❖ A bundle of Liferay Digital Experience Platform (bundled with Tomcat for JDK 8) can be found in the provided materials.
- ❖ The file looks like this:
`liferay-dxp-digital-enterprise-[version].zip`

EXERCISE: CREATING THE BUNDLE DIRECTORY STRUCTURE

1. **Create** the following directory structure:
 - Windows: C:\liferay\bundles
 - Mac/Linux: [user-home]/liferay/bundles
2. **Expand** the file.liferay-dxp-digital-enterprise-[version].zip in the bundles directory.
 - **NOTE:** Windows's built-in archive tool can take a long time to extract large .zip files. We recommend using a third-party tool, like 7-Zip or WinZip.

EXERCISE: LIFERAY-TOMCAT BUNDLE

1. **Go to** Tomcat's bin directory.
 - Windows: **Go to** C:\liferay\bundles\liferay-dxp-digital-enterprise-[version]\tomcat-[version]\bin using the file manager.
 - Mac/Linux: **Go to** [user-home]/liferay/bundles/liferay-dxp-digital-enterprise-[version]/tomcat-[version]/bin in a *Terminal* window.

EXERCISE: STARTING TOMCAT

1. **Start** Tomcat.

- Windows: **Double-click** on startup.bat.
 - You can also go to the command window, type catalina run, and press *Enter*.
- Mac/Linux: **Run** the ./catalina.sh run command in the *Terminal*.
 - **NOTE:** You'll need to make .sh files executable. Just type chmod a+x *.sh into the *Terminal*, from the Tomcat bin directory.

TIP FOR MAC USERS

- If you're not comfortable navigating and running commands through the *Terminal*, you can rename the startup.sh file in the bin/ folder to startup.command.
- This will make the file runnable, so you can just double-click on it in *Finder* and start Liferay.
- You can do the same thing to shutdown.sh, so you can easily start and stop Liferay.
- **NOTE:** If you do this, you won't be able to see console messages from Liferay. Liferay will run "hidden", but will otherwise function the same.

CHECKPOINT!

- ✓ You should see output as below in your console when Liferay starts successfully:

```
06-May-2016 17:10:40.105 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["http-nio-8080"]
06-May-2016 17:10:40.118 INFO [main] org.apache.coyote.AbstractProtocol.start
Starting ProtocolHandler ["ajp-nio-8009"]
06-May-2016 17:10:40.118 INFO [main] org.apache.catalina.startup.Catalina.star
Server startup in 92782 ms
```

EXERCISE: DEPLOYING THE ACTIVATION KEY

- Once the server starts up, Liferay will open our default browser to the Basic Configuration screen.
 - Before going through the basic configuration, we must add the license key to Liferay; otherwise, we'll hit an error message.
 - A directory named `deploy/` is created when we run Liferay.
1. **Copy** the `activation-key-developement-[version].xml` file that you received.
 2. **Paste** the file into the
`liferay-dxp-digital-enterprise-[version]/`
deploy folder.
 - You should see that the license is registered by looking for License registered for Portal Development in the Terminal/Console.
 - Once confirmed, we are able to proceed with our Basic Configuration without any errors.

EXERCISE: SETTING EVERYTHING UP

1. **Type S.P.A.C.E.** for the *Portal Name*.
2. **Uncheck** the *Add Sample Data* box.
3. **Type** the first name *Space*, the last name *Admin*, and the email address *space.admin@spaceprogram.liferay.com* for the Administrator User.
4. **Click** the *Finish Configuration* button to use the default HSQL database.

The screenshot shows the Liferay DXP Basic Configuration interface. On the left, there's a large blue circular logo. To its right, the text "Liferay DXP" is displayed. The configuration form is divided into two main sections: "Portal" and "Administrator User".
Portal:

- Portal Name: S.P.A.C.E.
For example, Liferay.
- Default Language: English (United States) (with a "Change" button).
- Add Sample Data

Administrator User:

- First Name: Space
- Last Name: Admin
- Email: space.admin@spaceprogram.liferay.com

Database:

Default Database (Hypersonic)
This database is useful for development and demo'ing purposes, but it is not recommended for production use. (Change)

Finish Configuration button at the bottom.

WWW.LIFERAY.COM

LIFERAY.

EXERCISE: READING THE TERMS OF USE

1. **Click** *I Agree* after you have read the *Terms of Use*.



Terms of Use

Welcome to our site. We maintain this web site as a service to our members. By using our site, you are agreeing to comply with and be bound by the following terms of use. Please review the following terms carefully. If you do not agree to these terms, you should not use this site.

1. Acceptance of Agreement

You agree to the terms and conditions outlined in this Terms of Use Agreement ("Agreement") with respect to our site (the "Site"). This Agreement constitutes the entire and only agreement between us and you, and supersedes all prior or contemporaneous agreements, representations, warranties and understandings with respect to the Site, the content, products or services provided by or through the Site, and the subject matter of this Agreement. This Agreement may be amended at any time by us from time to time without specific notice to you. The latest Agreement will be posted on the Site, and you should review this Agreement prior to using the Site.

2. Copyright

The content, organization, graphics, design, compilation, magnetic translation, digital conversion and other matters related to the Site are protected under applicable copyrights, trademarks and other proprietary (including but not limited to intellectual property) rights. The copying, redistribution, use or publication by you of any such matters or any part of the Site, except as allowed by Section 4, is strictly prohibited. You do not acquire ownership rights to any content, document or other materials viewed through the Site. The posting of information or materials on the Site does not constitute a waiver of any right in such information and materials.

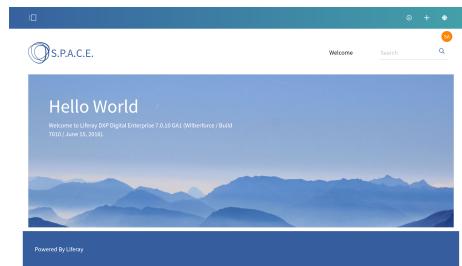
3. Service Marks

WWW.LIFERAY.COM

LIFERAY.

EXERCISE: FINISHING SET-UP

- ❖ Before you can start using Liferay, you have a couple more things to do:
 1. **Type** a new password (we typically set it to **test**).
 2. **Click Save**.
 3. **Choose** a password reminder query.
 4. **Click Save**.
- ✓ Now you're ready to go.



WWW.LIFERAY.COM

 LIFERAY.

WHAT HAPPENED BEHIND THE SCENES: LIFERAY HOME

- ❖ `LIFERAY_HOME` is the primary location for any custom Liferay properties or configuration files.
- ❖ In a production environment, this will almost always be configured to a specific location that is solely being used for configuration files.
- ❖ For our purposes in this class, we'll just be using the default location, which is in your `liferay-dxp-digital-enterprise-[version]` folder, one level in the directory structure above where the application server is located.

WWW.LIFERAY.COM

 LIFERAY.

WHAT HAPPENED BEHIND THE SCENES

- ❖ Liferay created a file called `portal-setup-wizard.properties` in `LIFERAY_HOME` to store the basic configuration properties.
- ❖ There are two ways to override properties: through Liferay's Control Panel or by creating a file called `portal-ext.properties` in `LIFERAY_HOME`.
- ❖ The properties defined in `portal-ext.properties` override properties in the `portal.properties` file located in `LIFERAY_HOME/tomcat-[version]/webapps/ROOT/WEB-INF/lib/portal-impl.jar`.
- ❖ Liferay reads the `portal.properties` file first, then the `portal-ext.properties` file, and finally the `portal-setup-wizard.properties` file.

EXERCISE: BRACKETS

- ❖ For this course, we will be using the Brackets text editor.
 - ❖ We'll need to install Brackets to use some snippets for later development.
1. Go to the *exercises/front-end-developer-exercises/o1-development-environment/brackets*.
 2. Double-click the installer for your operating system.
 3. Click through the installation steps.
- ✓ Now we have Brackets installed! We will use this later for development snippets.
- ❖ You can find more details at the Brackets website: <http://brackets.io/>

LIFERAY DEVELOPER NETWORK

- ❖ For more information on many of the topics we'll cover in this course, as well as resources for Liferay development and system administration, you can visit the Liferay Developer Network at dev.liferay.com.
- ❖ The Liferay Developer Network is home to the official documentation and user guide, tutorials, **Learning Paths** for developing on Liferay, and community-contributed resources like the Liferaypedia.



WWW.LIFERAY.COM

 LIFERAY.

Notes:



THE SPACE PROGRAM ACADEMY OF CONTINUING EDUCATION

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

S.P.A.C.E.

- ❖ It's 2065, and The Space Program has founded S.P.A.C.E. (The Space Program Academy of Continuing Education) with the interest of educating the next generation of innovators and astronauts. The most recent developments in space technology (e.g., *One-Man Pod* and *Recycled Rocket*) and the increasing popularity of space-related careers rendered S.P.A.C.E.'s foundation a necessity.
- ❖ Initial enrollment is far exceeding expectations. This has sent S.P.A.C.E. over the moon, but they're experiencing major growing pains and setbacks as their portal proves unable to handle their demands.

THE S.P.A.C.E. MISSION

- With so many students and a rapidly-expanding vision for the future, it's time for S.P.A.C.E. to give their old, defunct portal a front-end facelift.
- As part of the S.P.A.C.E Front End Development team, you are tasked with styling S.P.A.C.E.'s site and content.
- A senior S.P.A.C.E. and Liferay Instructor will guide you through this process. The skills you gain making S.P.A.C.E. beautiful will give you valuable experience so you can help your other projects take off.

THE BIG PICTURE: CHALLENGES

- The S.P.A.C.E. team has encountered a number of challenges dealing with their content and styling:
 - A theme that looks like it came from the 90's
 - Inconsistent and outdated information
 - Lackluster content
 - Application layouts that have caused poor user experience

THE BIG PICTURE: FRONT-END GOALS

- ❖ With these challenges in mind, S.P.A.C.E. has laid out some goals:
 - A custom theme that provides a responsive framework for content
 - A custom layout template for specific pages
 - Beautifully styled content for the front pages
 - Branded and styled email notifications
 - Styled applications for news and events feeds

S.P.A.C.E. RIGHT NOW

- ❖ Our admins have already begun setting things up, implementing tools such as Sites and Organizations.
- ❖ System Admins have been working on the infrastructure to bring in Users from another system.
- ❖ Content creators have been hard at work creating content in Liferay.

THE S.P.A.C.E. MISSION AND YOU

- As part of the S.P.A.C.E. team, we will work together to implement solutions in Liferay that best fit the needs of the Academy.
- We're responsible for executing on S.P.A.C.E.'s front-end goals. We have to make sure:
 - Our theme will work across different browsers
 - Layout templates will render our applications across different channels
 - Content is rendered properly
 - Styling does not interfere with content being displayed

Notes:

Chapter 2

New UI Landscape



LIFERAY'S UI TECHNOLOGIES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

BUILDING USER EXPERIENCE

- ❖ Liferay DXP provides a set of technologies to unify the user experience.
- ❖ Users expect a consistent experience from app to app.
- ❖ Liferay's UI is based on an *experience language*.
- ❖ This experience language is called *Lexicon*.
- ❖ *Lexicon* describes how a user interacts with Liferay.



WHAT IS AN EXPERIENCE LANGUAGE?

- Think back on some of the design in early smartphones.
- Were the icons always consistent with each other?
- Did the style of various widgets, like clocks or weather displays, mesh with the overall experience?



- Not so much.

A UNIFIED EXPERIENCE

- The goal of an experience language is to provide a unified experience through all aspects of a site or application.
- A simple example of this is button placement:
 - Consistently placing the “Next” or “OK” button in the bottom right-hand corner of a prompt, and placing the “Back” or “Cancel” button to the left would provide consistency in an interface and prevent a user from mistakenly cancelling out of a dialog that he or she wanted to continue.
 - Ensuring that not just the relative positioning, but also the absolute positioning is similar in each case would be a way to create an even more unified experience.
- Consistent colors and textures across an application are another aspect.
- Overall, the goal is to provide a visual design where everything flows and fits together, not unlike a spoken language.

BUILDING USER INTERFACES

- Lexicon can only describes what Liferay will look and feel like.
- To actually build these experiences, we have to use styles and behaviors.
- Liferay implements various technologies to build its user experience:
 - HTML, CSS, Images, Fonts
 - Lexicon CSS
 - JavaScript
 - Lexicon CSS
 - jQuery
 - Metal JS
 - AlloyUI
 - Templates
 - Web Content Templates
 - Application Display Templates
 - Soy/JSX Templates

EXPLORING THE TECHNOLOGIES

- We'll be using these technologies to help define the S.P.A.C.E. experience.
- Let's take a brief look at what these technologies are.
- We'll then dive into implementing a consistent user experience using these tools.

Notes:



INTRODUCING LEXICON CSS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

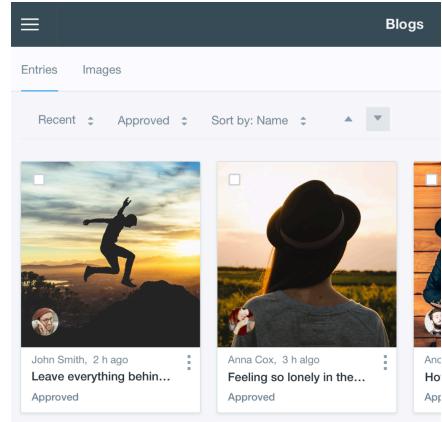
WHAT IS LEXICON CSS?

- ❖ Lexicon CSS is a web implementation of Liferay's Lexicon Design Language.
- ❖ Lexicon CSS provides style guidelines and best practices for designing web applications.
- ❖ It is meant to be fluid and extensible while providing a complete collection of visual and interactive patterns.
- ❖ It is an extension of the Bootstrap 3 Framework and is built with SASS.
- ❖ Lexicon CSS fills the front-end gaps between Bootstrap and the specific needs of Liferay.
- ❖ Lexicon CSS is optimized for the most common operations and takes the best from the most popular modern UI patterns available.
- ❖ These patterns can be carefully combined, like Lego pieces, to obtain the desired user experience.



LEXICON CSS FEATURES

- Lexicon CSS has components and features to cover most use cases:
 - Responsive layouts
 - Modern fonts
 - Aspect Ratio
 - Cards
 - Dropdown Wide and Dropdown Full
 - Figures
 - Nameplates
 - Sidebar/Sidenav
 - Stickers
 - SVG Icons
 - Timelines
 - Toggles



REUSABLE PATTERNS

- Lexicon CSS is built on reusable patterns to make common tasks easy:
 - Truncating text
 - Content filling the remaining container width
 - Truncating text inside table cells
 - Table cells filling the remaining container width and table cells only being as wide as their contents
 - Open and closed icons inside collapsible panels
 - Nested vertical navigations
 - Slideout panels
 - Notification icons/messages
 - Vertical alignment of content

BUILDING A THEME

- ❖ Most of our look and feel will come from a *theme*.
- ❖ Developers start with the *Lexicon Base* theme.
- ❖ *Lexicon Base* is our Bootstrap API extension and includes all of the default Lexicon CSS styles.
- ❖ You can then add styles and images to build your custom brand.

USING ATLAS

- ❖ Instead of building from scratch using *Lexicon Base*, you may want to start from a full theme.
- ❖ Atlas is Liferay's custom Bootstrap theme, and provides a solid base theme.
- ❖ Liferay uses Atlas to build our Classic theme that ships out of the box.
- ❖ You can use Atlas to build and modify our base theme in a reusable way.

USING LEXICON CSS

- Lexicon CSS can be used in both theme development and template development.
- Later, we'll create our own custom themes and templates that take advantage of these features.
- For some additional details, see the appendix *Appendix - Lexicon CSS Details*.
- For a list of components, elements, etc., visit:
<http://liferay.github.io/clay/>



Notes:



JAVASCRIPT IN DXP

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT DOES JAVASCRIPT LOOK LIKE IN DXP?

- The front-end of Liferay DXP is designed to be extensible, flexible, and feature-ready.
- As a platform, Liferay DXP is compatible with the more widely adopted front-end libraries, allowing developers to choose whatever framework suits their needs.
- Like previous versions, many components are written using *AlloyUI*.
 - AlloyUI is no longer actively maintained.
- Because of this, we have included *jQuery*, and have also developed a new framework called *Metal.js*.

JQUERY

- ❖ JQuery is a cross-platform JavaScript library designed to simplify the client-side scripting of HTML.
- ❖ It is the most popular JavaScript library in use today.
- ❖ JQuery's syntax is designed to make it easier to navigate a document, select DOM elements, create animations, handle events, and develop Ajax applications.
 - Although jQuery is great for small websites, once you start creating high, scalable applications like Java portals, you'll need a more robust solution.

ALLOYUI

- ❖ AlloyUI is an open-source front-end framework built on top of Yahoo! User Interface Library (YUI).
- ❖ It leverages all of YUI's modules and adds more components and features to help you build your UIs.
- ❖ AlloyUI is also server-agnostic, so you can use it with any technology.
- ❖ AlloyUI is deprecated in DXP but can be maintained and customized for legacy purposes.

METAL.JS

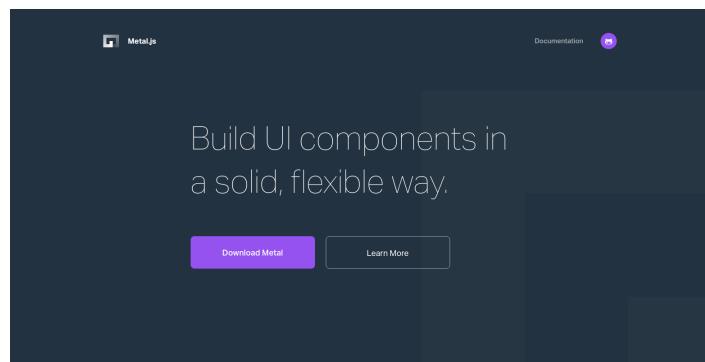
- ❖ Metal.js is a JavaScript library for building UI components in a solid, flexible way.
- ❖ Metal.js is built from the ground up with performance in mind.
- ❖ Metal.js is flexible enough to be built as global objects, AMD modules, or jQuery plugins.
- ❖ Metal.js uses the *ECMAScript 2015* language specification.

WWW.LIFERAY.COM

 LIFERAY.

METAL.JS EXAMPLE

- ❖ Internally, Liferay will be focusing on Metal.js as our primary JavaScript framework for front-end development.
- ❖ To see some examples and get more information, you can go to the Metal.js website: <http://metaljs.com/>



WWW.LIFERAY.COM

 LIFERAY.

Notes:



CUSTOM DISPLAYS WITH TEMPLATES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

FINISHING TOUCHES

- Using Liferay to build a user experience means using:
 - **Lexicon CSS** – provides HTML, CSS, and JavaScript for consistent design
 - **Metal.js and AlloyUI** – provides JavaScript components to enhance behavior and interaction
 - **Themes** – using technologies like Lexicon CSS to provide a base look and feel across Liferay
- We may also need to fine-tune the user experience in specific apps and content.

DEFINING THE CONTENT EXPERIENCE

- ❖ Liferay's content management provides a powerful way to communicate to users.
- ❖ Front-end developers can help direct the design and interaction with Web Content.
- ❖ *Web Content Templates* provide a direct way to define the user experience.

REFINING APP INTERACTION

- ❖ All of the content built will be displayed through different applications.
- ❖ Designers and developers can help shape the user interface for many applications.
- ❖ Lots of content-driven apps like the *Asset Publisher* can be modified.
- ❖ *Application Display Templates* allow us the freedom to control app experiences.

DEVELOPING MODERN APPS

- ❖ Building a highly responsive modern app means working closely with design.
- ❖ Liferay provides a framework to build modern apps.
- ❖ Metal.js apps can use powerful templates to build user interfaces.
- ❖ Developers and designers can work together using *Soy* and *JSX* templates to build new apps effectively.

PUTTING TECHNOLOGIES TO USE

- ❖ All of these technologies will be used to define the user interaction we want for S.P.A.C.E.
- ❖ We'll build a theme using technologies like *Lexicon CSS*, *AlloyUI*, and *Metal.js* to provide a look and feel for all of DXP.
- ❖ We'll look at template technologies like *Web Content Templates* and *Application Display Templates* to fine-tune the display on different types of content.
- ❖ More information on building modern UIs can be found in the appendix - *Appendix - Modern Application Design with Templates*.

Notes:

Chapter 3

Theme Development



STYLING LIFERAY WITH THEMES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

GETTING THE RIGHT LOOK

- No matter how beautiful the default look and feel is, you'll likely want to customize it to suit your needs.
- Liferay defines a consistent style that controls every part of the UI:
 - Buttons
 - Text
 - Icons
 - Page layout
- With a little effort, you can implement consistent branding across the platform.

THE S.P.A.C.E. CASE

- The situation at S.P.A.C.E. is no different.
- After setting up Liferay DXP, the design team wants to implement a stellar look and feel.
- They'll need to control all of the styles across Liferay:
 - Typeface
 - Buttons
 - Background color
 - Images
- They'll also want to modify some of the page structure:
 - HTML of the page
 - Layout of applications on the page

THEMES CAN DO IT

- Liferay uses *Themes* to control the user experience.
- Most of our needs are met with *Themes*. We can:
 - Control the HTML of every page
 - Control global styles for text, color, and more
 - Define the icon set
 - Set margins
 - Control CSS animations
- There's one thing we can't really do with *Themes*:
 - Layout of applications on the page
- We'll tackle this one later.

WHAT'S IN A THEME?

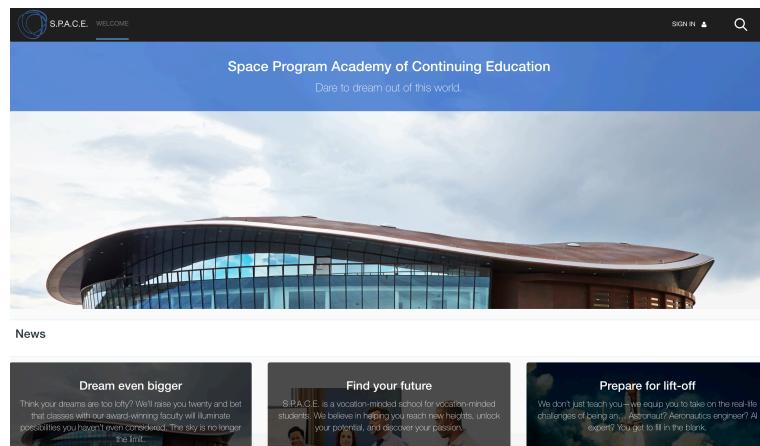
- ❖ A Liferay *Theme* is a collection of files:
 - ❖ CSS
 - ❖ JavaScript
 - ❖ Images
 - ❖ Templates
- ❖ These files are packaged up in a *module* that gets deployed in your instance of Liferay.

WWW.LIFERAY.COM

LIFERAY.

WHAT A THEME CONTROLS

- ❖ Here's a look at what the Theme controls:



- ❖ Note that this includes all the text, buttons, links, and other UI elements controlled by CSS.

WWW.LIFERAY.COM

LIFERAY.

WHAT'S NOT IN A THEME

- Your *Theme* defines the basic user experience in Liferay.
- More customization can be done in additional areas:
 - Menus and Panels
 - Layout of applications on the page
 - Styling for web content
 - Customizing application displays
- These can be customized through templates or custom modules.
- You'll probably need development help for custom modules.

WHAT DEVELOPERS CAN DO

- Developers can create modules to override:
 - The Product Menu
 - The Control Menu
 - Control Panel
 - Site Administration Panel
 - Simulation Menu
 - Some application displays

THEME ON

- We'll dig into some additional customizations in a bit.
- Our first problem to tackle is getting the basic look and feel of S.P.A.C.E. right.
- This means creating a custom *Theme*.

THEMATIC TOOLS

- Themes are modules that are a combination of CSS, JavaScript, HTML, and FreeMarker templates that modify the look and feel of Liferay.
 - At a high-level, themes in Liferay DXP haven't changed that much since Liferay 6.2.
- What has changed are the tools and libraries used in developing themes.
- Here is what we'll cover:
 - Liferay Theme Generator
 - Themelets
 - Bourbon
 - Theme Contributors
 - Importing Resources

THEME STRUCTURE

- ❖ Custom Themes are created based on one of two *base themes*.
- ❖ Liferay's base themes provide a starting look and feel to customize:
 1. **Unstyled:** The unstyled theme maintains basic functionality with no styling.
 2. **Styled:** The styled theme inherits from the unstyled theme, and simply adds some Lexicon CSS styling that inherits Bootstrap's styles.
- ❖ Using a base theme as your foundation, you can then make your customizations to the theme files.

LIFERAY THEME GENERATOR

- ❖ A suite of development tools is available for Liferay DXP to make theme creation easier.
- ❖ New themes can be built using the *Liferay Theme Generator* in Yeoman.
- ❖ The *Liferay Theme Generator* is a new tool for quick and easy development of Liferay themes.

THEMELETS

- ❖ Sometimes you don't need to override the entire look and feel.
- ❖ You may want to modify only a few areas of the theme:
 - Colors
 - Fonts
- ❖ This is easy to do with *Themelets*.
- ❖ Themelets are small, extendable, reusable pieces of code that are implemented by a theme.
- ❖ They can consist of CSS, templates, images, and JavaScript just like a theme.
- ❖ They are created as *npm* packages and can be published to the *npm* registry for easy sharing and reuse.

HANDLING CROSS-BROWSER COMPATIBILITY WITH BOURBON

- ❖ *Bourbon* is a lightweight mixin library for Sass, which is used when processing SCSS files, for Liferay DXP.
- ❖ It provides a number of mixins to handle CSS3 features.
- ❖ As an example, this:

```
section {  
  @include linear-gradient(to top, blue, black);  
}
```
- ❖ Translates to this:

```
section {  
  background-color: blue;  
  background-image: -webkit-linear-gradient(bottom, blue, black);  
  background-image: linear-gradient(to top, blue, black);  
}
```
- ❖ For more information on *Bourbon* examples, you can go here:
<http://bourbon.io/docs/>

THEME CONTRIBUTORS

- ❖ Additionally, developers can now create *Theme Contributors*.
- ❖ Theme Contributors provide a way to package UI resources independent of a theme and include them on the page.
- ❖ There are theme contributors that come packaged out of the box.
- ❖ These have UI resources that style the Product Menu, Control Menu, and Simulation Menu.

IMPORTING RESOURCES

- ❖ Lastly, as was true in previous versions, developers can package up and deploy content with their theme.
- ❖ The Resources Importer allows developers to deploy their themes with predefined content.
- ❖ This is useful for showcasing a theme and also provides a Site Template that can be used for creating new sites with a predefined look and feel.

LET'S DO IT!

- ❖ This may seem like a lot of info, but we'll look at each topic we mentioned one at time.
- ❖ We can use our new theme to start defining the S.P.A.C.E. user experience.
- ❖ We'll then build up more customizations on top.
- ❖ Ready? Let's build a theme!

Notes:



USING THE LIFERAY THEME GENERATOR

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

START YOUR THEMES

- We're ready to start defining the look and feel of S.P.A.C.E. using a theme.
- Defining the branding across Liferay will take a few steps:
 - Creating the theme
 - Customizing the HTML on the pages
 - Defining our styles in CSS
 - Customizing images
 - Adding any JavaScript we need
 - Finishing the theme module's configuration
- This should deal with most of what the design team needs to establish branding.

LIFERAY THEME GENERATOR

- We'll first need to create our theme using the installed tools.
- The two main tools are:
 - **Liferay Theme Generator:**
<https://github.com/liferay/generator-liferay-theme>
 - **Liferay Theme Tasks:** <https://github.com/liferay/liferay-theme-tasks>
- With these two tools, we'll speed up our development cycle of *create-modify-test*.

WHAT DO THE TOOLS DO?

- Liferay Theme Generator is a *Yeoman* generator that is used to create the theme boilerplate.
- Liferay Theme Tasks is a set of *gulp* tasks used for building and deploying your theme.
 - These tasks are automatically installed when creating a theme with Liferay Theme Generator.

YEOMAN GENERATORS

- There are four generators in the generator-liferay-theme package.
 - liferay-theme
 - liferay-theme:import
 - liferay-theme:layout
 - liferay-theme:themelet
- Here we will focus on liferay-theme and liferay-theme:import.

EXERCISE: CREATING THE SPACE THEME

- Let's create the starting point for our theme!
1. **Go to** the *liferay* folder you created in the command line or *Terminal*.
 - **Windows:** *C:\liferay*
 - **Mac/Linux:** *~/liferay*
 2. **Run** the *yo liferay-theme* command.
 - This command will create the base theme files and install the necessary dependencies for deployment.
 3. **Type** *Y* or *N* if prompted to collect data.
 4. **Press Enter.**

EXERCISE: FINISHING UP THE SPACE THEME CREATION

1. **Type** to name your theme *Space Program Theme*.
 - This is what administrators see when they look for your theme in Liferay.
2. **Press Enter**.
3. **Press Enter** to accept the default themId.
 - This is what Liferay calls your theme internally.
4. **Choose** *7.0* for the theme version.
 - Liferay DXP is built on Liferay 7.0.

EXERCISE: SELECTING AN APP SERVER

- Once you are done following the prompts, it will create a new directory containing the theme files and install the *npm* dependencies.
- Next, let's choose our app server directory:
 1. **Type** in the command line or *Terminal*.
 - Windows: *C:\liferay\bundles\liferay-dxp-digital-enterprise-[version]\tomcat-[version]*
 - Mac/Unix: *~/liferay/bundles/liferay-dxp-digital-enterprise-[version]/tomcat-[version]*
 - Make sure there are no spaces at the end of the path.
 2. **Press Enter** to use the default *http://localhost:8080* for the URL.
- ✓ We now have a base theme to customize!

NOT QUITE FROM SCRATCH

- We briefly explored that a new theme in Liferay is built from a *base* theme.
- This theme gives us a starting point with:
 - Basic structures
 - Basic styling
 - Complete UI components
- From here, we can add and customize to meet our vision.
- We can also build on top of other themes, not just the base theme.

LIFERAY THEME IMPORT

- If developers have an existing theme, which has not been setup to use Liferay Theme Tasks, they can use `yo liferay-theme:import`.
- This command does almost the same thing as `liferay-theme`, except it pulls in source files from an existing theme created using Plugins SDK.
- The only prompt will be for the root file path of the theme being imported. This generator will also create a new directory containing the theme files.



THEME STRUCTURE

- After running the `liferay-theme` generator, you will find the following file structure.

File	Description
<code>gulpfile.js</code>	Registers tasks from <code>liferay-theme-tasks</code> to your theme
<code>liferay-theme.json</code>	Generated file created by <code>gulp init</code> that stores appserver related configuration
<code>node_modules</code>	Directory where npm dependencies are installed
<code>package.json</code>	Where theme meta-data is defined and npm dependencies are declared
<code>src</code>	Contains source files of theme, similar to <code>_diffs</code> directory in Plugins SDK themes
<code>src/WEB-INF</code>	Meta-data files such as <code>plugin-package.properties</code> and <code>liferay-look-and-feel.xml</code>

DEPEND ON THESE MODULES

- You'll notice a file seen in other Node.js projects: `package.json`.
- This contains useful information:
 - Project metadata, such as name and description
 - Version number
 - List of dependencies
 - Node plugins
- If you open it, you'll notice dependencies like this:

```
"liferay-theme-deps-7.0": "*",
"liferay-theme-tasks": "*"
```
- Since the * indicates using the latest version, this theme will be built on newest dependencies.

EXERCISE: SIGNING INTO LIFERAY

- Let's deploy our base theme!
1. **Go to** *The Space Theme* using `cd space-program-theme` from your *liferay* directory in the command line or *Terminal*.
 2. **Run** the `gulp deploy` command.
 - This will make our base theme available on our Liferay instance.
 3. **Open** your browser.
 4. **Sign in** to Liferay.

EXERCISE: DEPLOYING OUR BASE THEME

1. **Go to** *Menu*→*Site Administration*→*Navigation* for the *S.P.A.C.E.* site.
2. **Click** *Options*→*Configure* next to *Public Pages*.
3. **Click** *Change Current Theme*.
4. **Choose** *Space Program Theme*.
5. **Click** *Save*.
6. **Click** *Go to Site* to see your theme.

BEHOLD THE SPACE PROGRAM THEME



S.P.A.C.E.

Breadcrumb

Breadcrumb

Welcome

Hello World

Welcome to Liferay DXP Digital Enterprise 7.0.10 GA1 (Wilberforce / Build 7010 / June 15, 2016).

Powered By [Liferay](#)

WWW.LIFERAY.COM

 LIFERAY.

ON THE SHOULDERS OF STYLED

- From here, we can start building our complete theme.
- Recall that the underlying base theme is *Styled*.
 - *Styled*: Contains the CSS, JavaScript, and everything needed to complete the base theme. *Styled* has a base theme of *Unstyled*.
 - *Unstyled*: Contains all the HTML structure and CSS class names.
- As we build, we'll be using gulp to perform some tasks.
- Let's get familiar with what they do.

WWW.LIFERAY.COM

 LIFERAY.

GULP TASKS: BUILD AND DEPLOY

- All gulp tasks are executed from the root directory of the theme.
- Here we will focus on the tasks that pertain to building and deploying your theme.

1. Build: `gulp build`

- This task compiles all source files into the `build` directory and creates a WAR file in the `dist` directory of your theme.

2. Deploy: `gulp deploy`

- The deploy task first runs the `build` task and then deploys the generated WAR file to the specified appserver.
- If your bundle is running, `deploy:gogo` can be used to deploy your theme. This method is faster than the traditional `deploy` task.
- **Note:** Only use one deploy method. If you first deploy with `deploy`, then don't switch to `deploy:gogo`, and vice versa.

GULP TASKS: WATCH AND INIT

3. Watch: `gulp watch`

- The `watch` task, similar to the `deploy:gogo` task, will only work when your bundle is running, as it communicates with the Gogo Shell.
- It watches for changes to theme source files and will “fast deploy” so that on page refresh your changes will take effect.

4. Init: `gulp init`

- The `init` allows you to specify your appserver path for use with the `deploy` task.
- It is also automatically invoked after `yo liferay-theme` and `yo liferay-theme:import`.
- These properties are saved in `liferay-theme.json`, which is created in your theme's root directory.

GULP TASKS: EXTEND

5. Extend: gulp extend

- » The extend task allows you to set what base theme you would like to extend your theme from.
- » It also lets you add themelets to your theme, which we'll get to later.

```
[10:26:01] Starting 'extend'...
? What kind of theme asset would you like to extend? (Use arrow keys)
> Base theme
  Themelet
```

- » To change the base theme, select Base Theme in the prompt.
- » The prompt will then let you select Styled or Unstyled.
- » The other two options allow you to extend from other themes, exposed as npm package, either installed on the system or published to the npm registry.
- » Styled and Unstyled are npm packages that have been published from liferay-portal source.

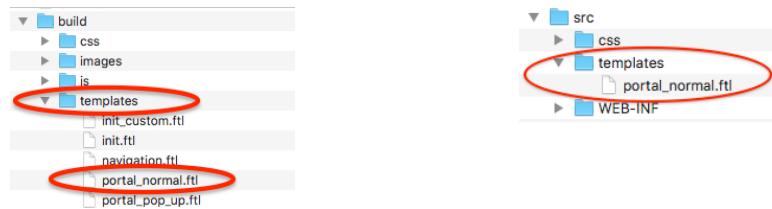
GULP TASKS: STATUS

6. Status: gulp status

- » The status task simply reports what base theme and themelets are implemented.

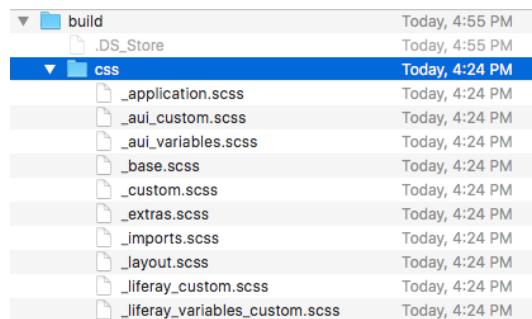
ADDING CUSTOMIZATIONS

- ❖ Customizing your theme's CSS, images, and templates is done in the `src/` folder.
- ❖ When you want to modify a file, originally from the base theme, you'll need to copy that file, from the `build` folder, to your `src` folder.
- ❖ Note: the `build` folder gets created after a `gulp build` or `gulp deploy`.



SASS EVERYWHERE

- ❖ Sass (SCSS) is the language Liferay uses for Stylesheets. Sass, combined with Bourbon's mixins, allows you to use the latest CSS3 features and leverage Sass's syntax advantages, such as nesting and variables.
- ❖ To make use of Sass's syntax and Bourbon, Liferay uses Sass to processes files with the `.scss` extension.

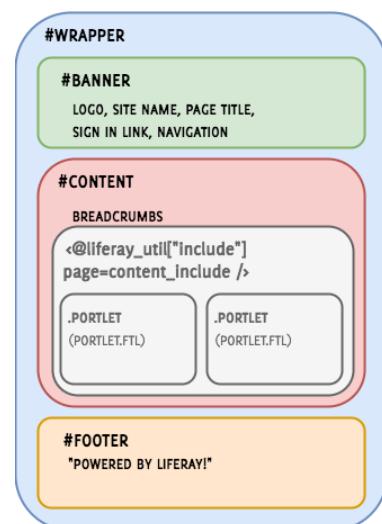


KEY FILES FOR THEME CUSTOMIZATION

- There are some key files to keep in mind when customizing your theme:
 - **portal_normal.ftl**: The main HTML source file
 - There are a number of HTML files we can modify, but the **portal_normal** file contains the main structure of the page.
 - **_custom.scss**: The file used for custom, global styling
 - Developers can style a number of elements in different files as seen in the build folder. These files are combined into one file called **main.css**, which is then linked into the theme's <head>, on every page.
 - **main.js**: Developers should place their custom JavaScript in this file, which is loaded on every page.
 - **liferay-look-and-feel.xml**: Developers can add different settings, such as *Theme Settings*, *Color Schemes*, and *packaged Layout Templates* in this file.

CONTROLLING PAGE STRUCTURE

- Themes have control over every aspect of the page.
- Every site page in Liferay breaks down to three major customizable sections as we see in the **portal_normal.ftl**:
 - **The Banner Section**: Includes the top part of a page with its sections
 - **The Content Section**: Includes breadcrumbs and code to render layouts and applications
 - **The Footer Section**: Includes the bottom of the page that can be customized



EXERCISE: MODIFYING OUR THEME

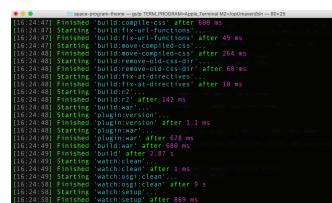
- We've provided all the necessary file changes for you, allowing us to dive right in.
- All of our theme exercise files can be found in `exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src`.
- First, we'll set up our theme's source folders.

1. **Go to** the Space Program Theme `src` folder.
2. **Create** the following folders in `src`:

- `fonts`
- `images`
- `js`
- `layouttpl`
- `templates`

EXERCISE: USING GULP WATCH FOR RAPID DEVELOPMENT

- Now that our theme is deployed and selected and we have the basic structure created, we can take advantage of gulp watch.
1. **Go to** your command line or *Terminal*.
 2. **Run** the `gulp watch` command.



```
13:57:32.471 Finished build:compile-css after 658 ms
13:57:32.471 Starting build:fix-uris-functions after 49 ms
13:57:32.480 Finished build:fix-uris-functions after 264 ms
13:57:32.480 Starting build:remove-old-css-dir after 60 ms
13:57:32.480 Finished build:remove-old-css-dir after 60 ms
13:57:32.480 Starting build:fix-at-directives after 10 ms
13:57:32.480 Finished build:fix-at-directives after 10 ms
13:57:32.480 Starting build:r2 after 142 ms
13:57:32.480 Finished build:r2 after 142 ms
13:57:32.480 Starting build:watch after 1 ms
13:57:32.480 Finished build:watch after 1 ms
13:57:32.480 Starting plugin:version after 1 ms
13:57:32.480 Finished plugin:version after 678 ms
13:57:32.480 Starting build:watch after 680 ms
13:57:32.480 Finished build:watch after 680 ms
13:57:32.480 Starting watch:clean after 1 ms
13:57:32.480 Starting Watch:osgi:clean after 1 ms
13:57:32.480 Starting Watch:setup after 5 s
13:57:32.480 Starting watch:setup after 659 ms
```

- Now we can see our changes as we make them.
- Check Liferay's log output to see when files have been deployed

COMING FROM LIFERAY 6.2

- If you've made themes in Liferay 6.2 or earlier, this looks really familiar!
- Some highlights of Liferay DXP themes versus Liferay 6.2 themes:
 - Themes were created and deployed using the Plugins SDK.
 - A Plugins SDK is still available, but newer tools with more features have been created to speed up development.
 - Liferay Theme Generator essentially accomplishes the same thing as the Plugins SDK's `themes/create.sh` script.
 - Unlike the Plugins SDK, you can create themes anywhere on your file system.
 - `gulp` is used for running build tasks instead of Ant.
 - Live updating of the theme is available through the `watch` task.
 - Building from existing themes is easier with the `extend` task.

THEME STRUCTURE CHANGES

- There are some specific changes in the structure and details of the theme module:
- All `.css` file extensions have been changed to `.scss`.
- Customizations go in a different folder structure:

<ul style="list-style-type: none">➤ Plugins SDK method<ul style="list-style-type: none">➤ Changes are in the <code>_diffs</code> directory.	<ul style="list-style-type: none">➤ Theme-Generator method<ul style="list-style-type: none">➤ Changes are in the <code>src</code> directory.
➤ FreeMarker templates are the recommended default:	
<ul style="list-style-type: none">➤ Liferay 6.2<ul style="list-style-type: none">➤ <code>portal_normal.vm</code>	<ul style="list-style-type: none">➤ Liferay DXP<ul style="list-style-type: none">➤ <code>portal_normal.ftl</code>

BUILDING THE THEME'S STRUCTURE

- With our basic theme module in place, we are free to add our custom brand.
- Inside the module's source folders, we'll be able to modify:
 - The HTML structure through templates
 - The global styling through CSS
 - Additional styling and branding through images
 - Custom JavaScript
- As we build each one, check on the progress in Liferay thanks to gulp Watch.

Notes:



CONTROLLING PAGE STRUCTURE

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

THE OUTLINE OF A THEME

- One of the first areas we'll deal with is the page structure.
- Just about every page in Liferay is generated from a template in the theme.
- If we want to control the HTML for:
 - The head
 - The body
 - The navigation
 - The footer
- Then we'll want to modify the templates in our theme.

WHAT WE NEED

- The S.P.A.C.E. design team has already outlined some changes we'll need to implement to apply our custom branding to Liferay.
- In addition to the main styling changes, we want to:
 - Customize the nav bar on the screen
 - Add a custom footer to the page
- As we get in there, we may find additional areas we want to customize.

FREEMARKER TEMPLATING LANGUAGE

- Chances are, you've worked with some templating system before.
- Templates have their own syntax, and generate HTML pages when processed.
- Your theme will use FreeMarker as the main templating language.
- As we go, keep an eye out for these features:
 - **Interpolations:** these are basic variable expressions that will be evaluated, and the result put on the page:
 `${default_url}`
 - **Directives:** special tags that tell FreeMarker to do something that doesn't always result in a visual difference:
 `<#assign default_url = "http://spaceprogram.liferay.com/" />`
- There are some other neat features, but we'll just focus on the basics for now.

WHAT MAKES UP A THEME?

- Let's take a look at what's in the theme and find our place:
 - **fonts** will contain any necessary custom fonts for the theme.
 - **images** will contain images that will be used in the theme.
 - **js** will contain JavaScript referenced in the themes templates.
 - **layouttpl** will contain any custom Liferay Layout Templates for the theme.
 - **templates** will contain the FreeMarker templates that provide the HTML structure of the theme.
- Let's walk through modifying some of the main files.
- We'll start by bringing in our theme files and modifying each section.

EXERCISE: SETTING UP BRACKETS SNIPPETS

- We've provided our theme's src files and snippets to walk through development.
- Let's start by setting up Brackets.
 1. **Open** Brackets.
 2. **Click** on the *Getting Started* dropdown in the top left menu.
 3. **Click** *Open Folder...*
 4. **Go to** *exercises/front-end-developer-exercises/o3-theme-development/o1-generating-a-theme/*.
 5. **Choose** the snippets folder.
- ✓ Now our snippets are ready! Let's get started.

TEMPLATES AT OUR DISPOSAL

- ❖ When an HTML page in Liferay is generated and sent to the browser, a few templates are used to create the source HTML:
 - `portal_normal.ftl`: This is the main HTML document for every page, containing the `<html>`, `<head>`, and `<body>` tags.
 - `portlet.ftl`: This contains the HTML that surrounds each application on the page.
 - This will be used when displaying applications with borders on the page, and will be inside the Layout Template. There is more on this in the Appendix.
- ❖ The main HTML document for all pages in Liferay can be pretty complex.
- ❖ We can include templates in other templates to modularize our page structure.
- ❖ For instance, we can separate our page navigation, body, and footer into different files.
- ❖ Let's set up our basic template files in the theme.

EXERCISE: MODIFYING OUR TEMPLATES

1. **Copy** the contents of the `exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src/templates` folder.
2. **Paste** the contents into the Space Program Theme's `src/templates` folder.
 - ❖ We've added a few template files for our theme:
 - `footer.ftl`: A custom ftl file we can use to separate the footer code from the main src file
 - `init_custom.ftl`: Includes our custom variables for use in the theme
 - `navigation.ftl`: Includes the HTML structure of the navigation in our theme
 - `portal_normal.ftl`: As mentioned before, this is our main source file where everything comes together.
 - `portlet.ftl`: This includes the HTML structure of application containers.

PAGE COMPOSITION

- We'll build the main page using two additional templates.
- This will make the structure easier to maintain and modify.
- Our main page will be laid out with `portal_normal.ftl`:
 - `HTML Head`
 - `JavaScript`
 - `Stylesheets`
 - `Body`
 - `Header`
 - `Navigation` - we'll include `navigation.ftl`
 - `Page layout` - controlled by a Layout Template
 - `Footer` - we'll include `footer.ftl`

EXERCISE: MODIFYING THE HTML SOURCE

- Let's walk through modifying our `portal_normal.ftl`.
- We'll add the header, body, and footer sections inside our wrapper `<div>`.

1. **Drop** the `portal_normal.ftl` file from your theme's `src/templates` folder into the Brackets editor.
2. **Open** the `templates` section under `snippets`.
3. **Click** on the `01-portal-normal-header` snippet.
4. **Copy** the contents of the snippet.
5. **Paste** the snippet contents over the `<!-- Insert snippet 01-portal-normal-header here -->` comment in the `portal_normal.ftl` file.

THE HEADER

- ❖ If you've worked with Bootstrap, the class names will look familiar.
- ❖ Our header is a `navbar`, with some other UI components included.
- ❖ In the Header section of the `portal_normal.ftl`, we've added a `fluid` class for responsive design.
- ❖ This section includes our navigation, site name, and logo.
- ❖ Our full navigation structure is in the `navigation.ftl` and is being included separately.

MAKING A STATEMENT

- ❖ You'll see all of our basic FreeMarker syntax in this template.
- ❖ Text and URLs that Liferay gives us are in variables:
 `${site_name}`
- ❖ We check to see if navigation is enabled using the `<#if />` directive:
`<#if has_navigation>
...
</#if>`
- ❖ We also check if we're supposed to show the Site name:
`<#if show_site_name>
...
</#if>`
- ❖ We'll see this kind of simple logic in most templates.

INCLUDING EXTERNAL TEMPLATES

- A useful directive for us is `<#include />`.
- This takes a template file name, and includes it in your template.
- It's the same as copying and pasting the contents:
 - FreeMarker looks in the same location as the current template.
 - If it finds the file, it adds the content of the template where the `<#include />` directive is.
 - The whole template file is processed, with this new template code in place.
- This makes separating our navigation into another file easy:

```
<#include "${full_templates_path}/navigation.ftl" />
```
- This also uses the `full_templates_path` variable Liferay gives to us.

USEFUL VARIABLES

- We're already seeing some helpful tools available in our theme templates.
- Some of the useful default variables include:
 - `site_name`: Returns the name of the Site, usually set by the Site Administrator
 - `site_default_url`: Returns the URL to the current Site
 - `site_logo`: Returns the URL to the Site logo image, usually set by the Site Administrator
 - `the_title`: The title of the application being displayed
 - `full_templates_path`: An absolute path on the file system to the folder containing the templates in the theme, useful for including external files.
- We'll see some more as we go.

EXERCISE: ADDING THE BODY SECTION

1. **Click** on the 02-portal-normal-main snippet.
2. **Copy** the contents of the snippet.
3. **Paste** the snippet contents over the <!-- Insert snippet 02-portal-normal-main here --> comment.
 - ❖ This section includes some accessibility classes.
 - ❖ This part of the template also includes an advanced feature: user-defined tags, or *macros*.
 - ❖ In the section, we have removed the breadcrumbs macro that's included as part of the navigation in the default portal_normal.ftl.

ADDING TEMPLATE FUNCTIONALITY WITH MACROS

- ❖ *Macros* allow you to assign template fragments to a variable, which makes them helpful for creating reusable pieces of template code.
- ❖ They will look very similar to normal tags:
`<@liferay.language_format />`
 - ❖ Instead of starting with a #, they begin with a @.
 - ❖ Like directives, they're used to perform different functions.
 - ❖ When the page is rendered, the macro is replaced with the template fragment.

DEFAULT LIFERAY MACROS

- ❖ Liferay provides a few default macros for integrating the most commonly used applications in your theme (e.g., search, breadcrumbs, user personal bar).
- ❖ There is also a set of default macros for users seeking to make their site more friendly to foreign languages.
- ❖ You'll find default Liferay macros used all around our templates.

LOOKING AT LIFERAY MACROS

- ❖ Let's take a look at some of the default macros:

Default Liferay Macros		
Macro	Parameters	Description
breadcrumbs	default preferences	Adds the Breadcrumbs portlet with optional preferences
control_menu	N/A	Adds the Control Menu portlet
css	filename	Adds an external stylesheet with the specified file name location
date	format	Prints the date in the current locale with the given format
js	filename	Adds an external JavaScript file with the specified file name source
language	key	Prints the specified language key in the current locale

ADDITIONAL LIFERAY MACROS

Default Liferay Macros		
Macro	Parameters	Description
language_format	arguments key	Formats the given language key with the specified arguments
languages	default preferences	Adds the Languages portlet with optional preferences
navigation_menu	default preferences instance ID	Adds the Navigation Menu portlet with optional preferences and instance ID
search	default preferences	Adds the Search portlet with optional preferences
user_personal_bar	N/A	Adds the User Personal Bar portlet

EXERCISE: FINISHING WITH THE FOOTER

- Let's finish up with the footer section.
 1. **Click** on the 03-portal-normal-footer snippet.
 2. **Copy** the contents of the snippet.
 3. **Paste** the snippet contents over the <!-- Insert snippet 03-portal-normal-footer here --> comment right under the </main> tag.
 4. **Save** the file.
 - In this case, instead of adding our footer code in portal_normal.ftl, we've created footer.ftl file that we're adding.
 - You'll recognize the <#include /> directive here.
 - The benefit of this approach is keeping the portal_normal.ftl file uncluttered.

EXERCISE: ADDING THE FOOTER.FTL

- ❖ Now that our `portal_normal.ftl` is referencing the `footer.ftl` file, we need to add our footer code.
 - 1. **Drop** the `footer.ftl` file from your theme's `src/templates` folder into the Brackets editor.
 - 2. **Click** on the `04-footer` snippet.
 - 3. **Copy** the contents of the snippet.
 - 4. **Paste** the snippet contents over the `<!-- Insert snippet 04-footer here -->` comment in the `footer.ftl` file.
 - 5. **Save** the file.
- ❖ Here we've added responsive fluid classes and a footer navigation menu.
 - ❖ You'll notice this is similar to the header, but with different content.

FOOTER DETAILS

- ❖ A few new directives show up in the footer:
- ❖ `<#assign />` lets us assign a value to a variable.
- ❖ Here, we're using it to call methods on objects, and store the result:
`<#assign VOID = freeMarkerPortletPreferences.setValue(...) />`
- ❖ When the `setValue()` method returns, that value is stored in `VOID`.
- ❖ We don't use `VOID` anywhere on the page, so that value never gets displayed.
 - ❖ This is a nifty way to keep return values from being added to the page.
- ❖ `<@liferay.navigation_menu />` is a macro that adds the Navigation application to the page.

FILLING IN THE FRAMING

- With our basic templates in place, we've now provided the basic structure for styling.
- After modifying the core `portal_normal.ftl`:
 - We have a custom navigation that complements our branding
 - We have a custom footer section in place that we can easily modify later
 - We can add or remove some of the basic features using macros
- We can now start applying our custom styles to make the new page structure fit our vision.

Notes:



STYLING THE PLATFORM

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

STYLING WITH PANACHE

- Once we've built our basic HTML structures through templates, we're ready to set the overall style of Liferay.
- Remember that themes control the basic look and feel across Liferay.
- That means all of our CSS styles will set the tone for our applications and content:
 - Typeface
 - Margins
 - Buttons and links
 - Background colors
 - Foreground colors
 - Highlight and accent colors
- Though we're moving through structure, styling, and configuration one at a time, you'll probably revisit each of these areas many times while putting the finishing touches on your theme.

DEFINING OUR BRAND

- ❖ The S.P.A.C.E. brand is defined by a number of items:
 - Typeface
 - Background color
 - Foreground color
 - Secondary and accent colors
 - UI element styles
 - Application border styles
- ❖ Many of these branding elements will be contained in our theme's CSS.
- ❖ For instance, our custom typeface will be a new font added to the theme.
- ❖ Others we can do using custom images.
- ❖ Some advanced customizations we'll use other modules or templates for.

GETTING SASSY

- ❖ All of Liferay's CSS is built on Sass.
- ❖ Sass (for Syntactically Awesome Style Sheets) adds extra features to CSS:
 - Variables
 - Mixins
- ❖ You may already be familiar with Sass, or similar technologies like Less.js.
- ❖ Some key syntax to look for:
 - *Variables* are written like this: \$variable
 - *Mixins* are written like this: @include mixin-name(...)
 - You may recall Bourbon – it has lots of mixins for us to use.
- ❖ So you may see stuff like this in our CSS:

```
@include linear-gradient(to top, gray, $brand-color);
```

EXERCISE: ADDING CUSTOM FONTS

- ❖ We want to use the *Open Sans* font in our theme.
 - ❖ Adding fonts into the theme is as simple as adding the font files and then referencing the fonts in an SCSS file.
 - ❖ Let's start by adding the font files to our theme.
1. **Go to** the exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src folder.
 2. **Copy** the files from the fonts folder.
 3. **Paste** the files into our new Space Program Theme `src/fonts`.
- ✓ Now we have fonts we can import!

THE CSS FOLDER STRUCTURE

- ❖ You can customize every styled aspect of the platform in a theme.
- ❖ The default `src/css` in build includes every `.scss` and `.css` file.
- ❖ Our theme includes customizations to some main SCSS files and breaks up other customizations into a `partials` and `portlet` folder.
- ❖ The `partials` folder includes custom styles for components while the `portlet` folder includes custom styles for our applications.
- ❖ We'll walk through adding the styling for some of these sections.



EXERCISE: MODIFYING WITH STYLE

- Next, let's add our CSS structure and walk through our styling.
- First, we'll add our CSS files to modify, and then we'll break down our file structure.

1. **Go to** the exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src folder.
2. **Copy** the files from the `css` folder.
3. **Paste** the files into our new Space Program Theme `src/css`.
 - This will have you replace the default `_custom.scss` in the theme.

EXERCISE: IMPORTING OUR FONTS

- Next, let's add our CSS structure and walk through our styling.
- First, we'll import the fonts we added earlier. This will allow us to use the fonts in our SCSS files and see changes on-the-fly with gulp watch.

1. **Drop** the `_fonts.scss` file from your theme's `src/css/partials` into the Brackets editor.
2. **Open** the `css` section under *snippets*.
3. **Click** on the `01-fonts-scss` snippet.
4. **Copy** the contents of the snippet.
5. **Paste** the snippet contents over the `// Insert snippet
01-font-scss` here comment in the `_fonts.scss` file.
6. **Save** the file.

ADDING FONTS EASILY

- ❖ Our font SCSS file uses *mixins* to simplify adding fonts:

```
@include font-face(...);
```

- ❖ The *font-face* mixin gives us a quick way to set a bunch of font values in one method.

- ❖ For instance, including a new font can be this easy:

```
@include font-face("open_sansregular", "../fonts/opensans-bolditalic-webfont",
```

- ❖ And will generate full CSS, something like:

```
@font-face {  
    font-family: "open_sansregular";  
    font-style: italic;  
    font-weight: bold;  
  
    src: url("../fonts/opensans-bolditalic-webfont");  
}
```

- ❖ Other mixins are used in similar ways — keep an eye out for them!

TAKING ADVANTAGE OF ATLAS STYLES

- ❖ As mentioned earlier, generated themes default to using the *Lexicon Base* theme.
- ❖ To include additional styles that are used in the classic theme, we can modify our *aui.scss* file to include the *Atlas theme*.
- ❖ The *Atlas theme* provides more styling than is included in the *_styled* theme.
- ❖ *Atlas* is Liferay's custom Bootstrap theme that is used in the Classic Theme.
- ❖ It overwrites and manipulates Bootstrap and *Lexicon Base* to create the classic Liferay look and feel.
- ❖ It is equivalent to installing a Bootstrap third party theme.

EXERCISE: UPDATING THE AUI.CSS FILE

- Let's modify the `aui.scss` to include the *Atlas theme*.
 1. **Drop** the `aui.scss` file from your theme's `src/css` into the Brackets editor.
 2. **Click** on the `02-aui-scss` snippet.
 3. **Copy** the contents of the snippet.
 4. **Paste** the snippet contents over the `// Insert snippet
02-aui-scss here` comment in the `aui.scss` file.
 5. **Save** the file.

CUSTOMIZING ALL THE SCSS

- The main file for adding any custom style changes is the `_custom.scss` file.
- This is where we can add our global styling for things like:
 - Background color
 - Accent colors
 - Applications Styling
 - Etc.
- The majority of the changes that S.P.A.C.E. requires will go in this file.
- Let's start by adding some basic customization to the file.

EXERCISE: ADDING CUSTOM STYLES

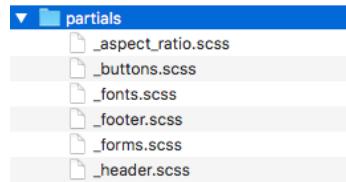
1. **Drop** the `_custom.scss` file from your theme's `src/css` into the Brackets editor.
2. **Click** on the `03-custom-scss` snippet.
3. **Copy** the contents of the snippet.
4. **Paste** the snippet contents over the `// Insert snippet
03-custom-scss here` comment in the `_custom.scss` file.
5. **Save** the file.

KEEPING THINGS ORGANIZED

- ❖ Now we have some general style changes we would like to make to the background and our applications.
- ❖ Based on the requirements, we still need to customize a number of other things on the platform.
- ❖ We could add all of the style changes we need to make here, but it could potentially get a bit unwieldy with hundreds of lines of code.
- ❖ What would be a good approach for keeping things organized and clean?
- ❖ This is where a more modular approach comes in to play.

MODULARITY WITH STYLE

- ❖ We've added the *partials* folder to organize each CSS component we want to customize.
- ❖ Each aspect of the platform we want to change will be its own file.
- ❖ For example, we want to change the button styles in our theme, so we have a *_buttons.scss* file in the partials folder.
- ❖ All we need to do from there is import the files into our *_custom.scss* file.
- ❖ This will make maintenance and organization much better for any theme scss in the future.



EXERCISE: ADDING THE PARTIALS IMPORTS

- ❖ Let's go ahead and import the partials styling into our *_custom.scss*.
1. **Click** on the 04-imports snippet.
 2. **Copy** the contents of the snippet.
 3. **Paste** the snippet contents over the // Insert snippet 04-imports here comment at the top of the *_custom.scss* file.
 4. **Save** the file.
- ✓ Now that they're all imported into the *_custom.scss*, we can add some styling.

INHERITANCE

- ❖ Our `_custom.scss` is a good place to some other useful SASS features.
- ❖ Sometimes we need to apply styles to areas of the DOM that are logically related.
- ❖ For instance, a span inside a div, all wrapped in another div.
- ❖ It'd be nice to show this hierarchy.
- ❖ Sass lets you do this:

```
.portlet-layout {  
    &.row {  
        margin: 0;  
        .col-md-12 {  
            padding: 0;  
        }  
    }  
}
```

- ❖ Instead of `.portlet-layout.row`, we can use `&` to show the relationship.

EXERCISE: STYLING THE FOOTER

- ❖ As an example of the more modular styles, let's add some styling to our `_footer.scss`.
- ❖ This file is being imported in our `_custom.scss` file and will include CSS for the `footer.ftl` we added earlier.

1. **Drop** the `_footer.scss` file from your theme's `src/css/partials` into the Brackets editor.
2. **Click** on the `05-footer-scss` snippet.
3. **Copy** the contents of the snippet.
4. **Paste** the snippet contents over the `// Insert snippet
05-footer-scss here` comment in the `_footer.scss` file.
5. **Save** the file.

EXERCISE: COLORFUL VARIABLES

- In some of our other .scss files, we have been referencing some color variables.
 - Having a color scheme represented as scss variables can simplify color styling throughout the theme.
 - Let's go ahead and set up our color variables.
1. **Drop** the _color.scss file from your theme's src/css/partials/variables into the Brackets editor.
 2. **Click** on the 06-colors-scss snippet.
 3. **Copy** the contents of the snippet.
 4. **Paste** the snippet contents over the // Insert snippet 06-colors-scss here comment in the _colors.scss file.
 5. **Save** the file.

EXERCISE: MAKING COLORS MODULAR

- Although it is possible to place any custom variables in the _variables.scss file, we have instead added a variables folder to keep our variables separate.
 - Since our color variables are the only variables added, we'll just import them into our _variables.scss file.
1. **Drop** the _variables.scss file from your theme's src/css/partials into the Brackets editor.
 2. **Click** on the 07-variables-scss snippet.
 3. **Copy** the contents of the snippet.
 4. **Paste** the snippet contents over the // Insert snippet 07-variables-scss here comment in the _variables.scss file.
 5. **Save** the file.

MODIFYING DEFAULT STYLES

- ❖ The base build folder includes a number of default styles that can be modified.
- ❖ These sections include things like application, navigation, layout styles, and more.
- ❖ This folder also includes the portlet folder with all the default styles.
- ❖ To modify application styles in a theme, you can simply copy that folder and the relevant .scss files into your theme's src folder.
- ❖ We already have the folder, so we can add some variable modifications for our applications.

EXERCISE: MODIFYING APPLICATION CSS

1. **Drop** the _variables_custom.scss file from your theme's src/css/portlet into the Brackets editor.
2. **Click** on the 08-portlet-variables-custom-scss snippet.
3. **Copy** the contents of the snippet.
4. **Paste** the snippet contents over the // Insert snippet 08-portlet-variables-custom-scss here comment in the _variables_custom.scss file.
5. **Save** the file.

EXERCISE: PROVIDING APPLICATION DECORATOR STYLES

- ❖ Application decorators provide borders for all the applications on a page.
 - ❖ We'll provide styling for the default application decorators in our theme.
 - ❖ We'll discuss application decorators in more detail in a later section.
1. **Drop** the `_portlet_decorator.scss` file from your theme's `src/css` into the Brackets editor.
 2. **Click** on the `09-portlet-decorators.scss` snippet.
 3. **Copy** the contents of the snippet.
 4. **Paste** the snippet contents over the `// Insert snippet
09-portlet-decorators.scss here` comment in the `_portlet_decorator.scss` file.
 5. **Save** the file.
 - ✓ If `gulp watch` is running, the theme will automatically update, if not, run the `gulp deploy` command.

WWW.LIFERAY.COM



SASS COMPLETE

- ❖ Our theme contains a number of other SCSS files for different aspects of the page.
- ❖ We've provided the rest of the styles so we can focus on adding the rest of our theme content.
- ❖ Now that we have our base html structure and styles down, let's add our theme images, JavaScript, and configurations.



Hello World

Welcome to Liferay DXP Digital Enterprise 7.0.10 GA1 (Wilberforce / Build 7010 / June 15, 2016).

WELCOME

WWW.LIFERAY.COM



EVEN MORE SASS

- There's a lot you can do with Sass and Bourbon.
- For more information on Sass, check out their site:
 - <http://sass-lang.com>
- Additional features, like a full-featured mixin library, are provided by Bourbon.
- To learn how to use all the advanced tools available through Bourbon, check out:
 - <http://bourbon.io>
- S.P.A.C.E. only uses a small portion of these features in the theme.

EXERCISE: ADDING IMAGES

- Let's add images first.
 - In general, we're going to keep any major images out of our theme.
 - Any images needed for a banner or footer content can be added later by a content team.
 - Our theme only needs a couple of images to provide a screenshot and thumbnail for use on the platform.
1. **Go to** the [exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src](http://www.liferay.com/learn/liferay-exercises/03-theme-development/01-generating-a-theme/exercise-src) folder.
 2. **Copy** the files from the images.
 3. **Paste** the files into our new Space Program Theme `src/images`.

THE REAL FAVICON

- ❖ We can also easily add or modify the *favicon* of the theme.
- ❖ *Favicons* (or **favorite icons**) are commonly used to make your website's brand recognizable as a small icon. Usually seen on bookmarks, desktop shortcuts, browser tabs, and home screen icons on a mobile device.



- ❖ Favicons can be stored as *ICO*, *JPG*, or *PNG* files.
- ❖ Sizes range from squares of **16x16** and **32x32** up to **310x310** pixels.

EXERCISE: ADDING FAVICONS TO YOUR THEME

- ❖ You can modify your favicon very easily by including a `favicon.ico` file in the `src/images` folder of the theme.
- ❖ Favicons can be generated on a number of websites, such as <http://realfavicongenerator.net/>
- ❖ Let's add a S.P.A.C.E. favicon to our theme.

1. **Copy** the `favicon.ico` from the `exercises/front-end-developer-exercises/o3-theme-development/o1-generating-a-theme` directory.
 2. **Paste** the `favicon.ico` in the space theme's `src/images` folder.
- ✓ Now we will see our favicon logo!



PAINTING THE PICTURE

- With our styling set using CSS, we have the overall look and feel of the S.P.A.C.E. platform defined.
- Custom images help enhance our brand identity, and give a place to put custom design work.
- Additional little details like Favicons can really tighten up the overall presentation.
- Even though we've tackled one of the biggest visual impacts of the theme, we'll still need look at:
 - Implementing custom JavaScript
 - Finish the theme's configuration

Notes:



ADDING CUSTOM JAVASCRIPT

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

NEW AND IMPROVED BEHAVIOR

- One of the core aspects to user experience is how the controls interact with the user.
- These can be changed through:
 - CSS
 - JavaScript
- Once our CSS has been dealt with, we can customize JavaScript easily through the theme.

GLOBAL JAVASCRIPT

- ❖ Since the theme controls the overall HTML of every page and the basic look and feel, it also defines the essential JavaScript libraries.
- ❖ If there are functions you need defined on each page or functions you need available globally, this is a great place to put them.
- ❖ You may also be building on different JavaScript libraries from the ones Liferay contains.
- ❖ Themes are a good place to include additional third-party JavaScript dependencies you may need for your applications and custom UI.
- ❖ S.P.A.C.E. is built on Liferay's default libraries, including:
 - AlloyUI
 - Metal.js
 - Lexicon CSS
- ❖ You can easily experiment and use your own libraries here.

EXERCISE: ADDING JAVASCRIPT

- ❖ Our theme needs to include a sign-in modal and some JavaScript for our top search.
- ❖ Let's add some JavaScript.
 1. **Go to** the [exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src](https://github.com/liferay/learn-exercises/tree/main/exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src) folder.
 2. **Copy** the files from the `js` folder.
 3. **Paste** the files into our new Space Program Theme `src/js`.

✓ Now we can modify our `main.js` file.

 - ❖ If the *Working Files* view in Brackets is getting cluttered, feel free to close the `.ftl` and `.scss` files.

EXERCISE: MODIFYING THE MAIN.JS

- » Let's add our sign-in modal using AUI.

1. **Drop** the `main.js` file from your theme's `src/js` into the Brackets editor.
2. **Open** the `js` section under *snippets*.
3. **Click** on the `01-main-js` snippet.
4. **Copy** the contents of the snippet.
5. **Paste** the snippet contents over the `// Insert snippet
01-main-js here` comment in the `main.js` file.
6. **Save** the file.

ALLOYUI, YUI, AND JQUERY

- » If you're familiar with YUI or jQuery, `main.js` looks pretty familiar.
- » AlloyUI is available through `AUI()`, much like `YUI()` or `$()`.
- » Just like a jQuery page might have:

```
$(document).ready(function () {
    // Do stuff...
});
```
- » AlloyUI (much like its parent YUI) uses:

```
AUI().ready(function (A) {
    // Do stuff...
});
```
- » You can see syntax comparison at the AlloyUI Rosetta Stone:
 - » <http://alloyui.com/rosetta-stone/>
- » While AlloyUI and jQuery are available if you want to use them, you can also include any other JavaScript framework in your theme.

METAL.JS AND ECMASCIPT 2015

- ❖ Along with AlloyUI, the S.P.A.C.E. theme is also built with another of Liferay's default libraries: Metal.js.
- ❖ As mentioned in a previous module, Metal.js is a JavaScript library that uses the ECMAScript 2015 language specification.
- ❖ By taking advantage of ECMAScript 2015 features like classes, modules and arrow functions, we can build modern, flexible UI components.

LIFERAY THEME ES2015 HOOK

- ❖ Our main.js requires the top_search.es.js file.

```
require(
  'space-program-theme/js/top_search.es',
  function(TopSearch) {
    new TopSearch.default();
  }
);
```

- ❖ The top_search.es.js uses Metal.js and ECMAScript 2015 syntax and will need to be transpiled to ensure that different browsers can read our theme.
 - ❖ Earlier, we discussed ECMAScript 2015 and transpilation - compiling source to be output in a different language from the input language.
 - ❖ Liferay provides an ECMAScript 2015 hook that has exactly what we need.

EXERCISE: LIFERAY THEME ES2015 HOOK

- ❖ The `liferay-theme-es2015-hook` allows for ECMAScript 2015 transpilation as some browsers haven't fully implemented all ECMAScript 2015 features yet.
- 1. **Open** the Terminal or your command line window.
- 2. **Go to** the root folder for your theme.
- 3. **Run** the `npm i -save liferay-theme-es2015-hook` command.
- ❖ Once the hook is installed, it will run with every `gulp build` and `gulp deploy`.
- ❖ After building, components will be transpiled and packaged as AMD modules.

EXERCISE: METAL.JS DEPENDENCIES

- ❖ As our `top_search.es.js` component uses the Metal.js framework, we'll also need to download the Metal.js dependencies to build properly.
- 1. **Open** the Terminal or your command line window.
- 2. **Go to** the root folder for your theme.
- 3. **Run** `npm i -save metal metal-dom metal-state`.

EXERCISE: UPDATING THE PACKAGE JSON

- Finally, we need to add the `hookModules` property to our `package.json` file to make sure our theme reads the hook during the build process.

1. **Open** the `package.json` file using Brackets.
2. **Type** `"hookModules": ["liferay-theme-es2015-hook"]`, in the `liferayTheme` section.
 - The `liferayTheme` property can be found on line 8.
3. **Save** the `package.json` file.

```
    "liferayTheme": {  
        "baseTheme": "styled",  
        "hookModules": ["liferay-theme-es2015-hook"],  
        "screenshot": "",  
        "rubySass": false,  
        "templateLanguage": "ftl",  
        "version": "7.0"  
    },
```

- ✓ Note: The search addition may take a few minutes to display after deploying the changes.

IMPORTING IN METAL.JS

- Let's take a look at some of the Metal.js code that features ECMAScript 2015 syntax.
- In the `top_search.es.js` file, we are creating the `TopSearch` class.
 - This class will create a basic component that enhances the default behavior of the search application form.
- We can take advantage of modules in ECMAScript 2015 that give us the ability to create, load, and manage dependencies via the new `import` and `export` keywords.

```
import async from 'metal/src/async/async';  
import core from 'metal/src/core';  
import dom from 'metal-dom/src/dom';  
import State from 'metal-state/src/State';
```

CLASSES IN METAL.JS

- ❖ Using ECMAScript 2015 features, we can also create classes with constructors and inheritance.

```
class TopSearch extends State {

    constructor() {
        super();

        this.search_ = dom.toElement('#search');
        this.searchIcon_ = dom.toElement('#banner .btn-search');
        this.searchInput_ = dom.toElement('#banner .search-input');

        ....
    }

    ...
}
```

WRITING FUNCTIONS WITH METAL.JS

- ❖ Arrow functions make creation of anonymous functions easier.
- ❖ With let, we can create variables with scope limited to the block in which they are declared.

```
onSearchInputBlur_(event) {
    async.nextTick(
        () => {
            let stateActiveElementBlur = document.activeElement !== this.searchIcon_
                && document.activeElement !== this.searchInput_;

            if (stateActiveElementBlur && (!this.searchInput_.value ||
                this.searchInput_.value === '')) {
                this.visible = false;
            }
        });
}
```

- ❖ With Metal.js, we can use let and other modern features to create powerful UI components.

OWNING YOUR JAVASCRIPT

- ❖ Themes are really useful ways to set the style and behavior across all of Liferay:
 - HTML structure
 - CSS styles
 - JavaScript libraries and UI components
- ❖ Once you've customized the JavaScript and added your own JavaScript components and libraries, you're ready to package up the theme.

Notes:



CONFIGURING THE THEME

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

FINISHING THE THEME

- ❖ Each of our basic areas for branding are defined:
 - HTML structure and basic navigation
 - Fonts, colors, and other styles in CSS
 - Behavior and UI libraries through JavaScript
- ❖ To finish our theme, we'll need to set some basic metadata that describes our theme module.
- ❖ Remember, we'll be deploying this branding as a module file in Liferay DXP.
- ❖ There are some advanced modifications we may also want to make.
- ❖ Some settings are easy, and can be done right now.
- ❖ Other changes we'll explore in more depth.

EXERCISE: ADDING LAYOUTS

- ❖ As we'll see later, you can create custom layout templates.
 - ❖ Layout templates can be bundled into our theme just like our fonts.
 - ❖ Let's start by including the layout template files, and then finish our theme's configuration.
1. **Go to** the www.liferay.com *exercises/front-end-developer-exercises/03-theme-development/01-generating-a-theme/exercise-src* folder.
 2. **Copy** the /custom folder from the layouttpl folder.
 3. **Paste** the folder into our new Space Program Theme src/layouttpl.
- ❖ We've included the layouttpl files and images.

FINISHING UP WITH THE THEME CONFIGURATION

- ❖ The last thing we need to do is provide special settings for our theme as well as some package properties.
- ❖ These configuration settings are found in the WEB-INF folder of the theme.
- ❖ The `liferay-look-and-feel.xml` file includes the configuration for things like custom layout templates, theme settings, and application decorators.
- ❖ The `liferay-plugin-package.properties` will contain the information and properties for our theme.

EXERCISE: ADDING OUR CONFIGURATION FILES

1. **Go to** the <exercises/front-end-developer-exercises/o3-theme-development/01-generating-a-theme/exercise-src> folder.
 2. **Copy** the files from the WEB-INF.
 3. **Paste** the files into our new Space Program Theme src/WEB-INF.
 - Replace both files.
- ✓ Now we can modify our configuration files!

ADDING APPLICATION DECORATORS

- In previous versions of Liferay, administrators could display or hide the application borders through the Show Borders option of the look and feel configuration menu.
- In Liferay DXP, this option has been replaced with Application Decorators, a more powerful mechanism that customizes the style of the application wrapper.

OUT OF THE BOX APPLICATION DECORATORS

- ❖ There are three out of the box application decorators that are added to your themes `liferay-look-and-feel.xml`:
 - **Barebone**: when this decorator is applied, neither the wrapping box nor the custom application title is shown. This option is recommended when you only want to display the bare application content.
 - **Borderless**: when this decorator is applied, the application is no longer wrapped in a white box, but the application custom title is displayed at the top.
 - **Decorate**: this is the default Application Decorator when using the Classic theme. When this decorator is applied, the application is wrapped in a white box with a border and the application custom title is displayed at the top.

EXERCISE: APPLICATION DECORATOR CONFIGURATION

- ❖ Let's start by adding our application decorator configuration.
1. **Drop** the `liferay-look-and-feel.xml` file from your theme's `src/WEB-INF` into the Brackets editor.
 2. **Open** the `WEB-INF` section under *snippets*.
 3. **Click** on the `01-portlet-decorators` snippet.
 4. **Copy** the contents of the snippet.
 5. **Paste** the snippet contents over the `<!-- Insert snippet
01-portlet-decorators here -->` comment in the `liferay-look-and-feel.xml` file.
 6. **Save** the file.

ADDING CUSTOM DECORATORS

- ❖ All the decorators added to the `liferay-look-and-feel.xml` require that a css class is set using the following line:

```
<portlet-decorator-css-class>[Class Name]</portlet-decorator-css-class>
```

- ❖ You can then add styling to these classes in the `_portlet_decorator.scss`.

```
.portlet-trending .portlet-content {  
    background-color: $brand-bg-color-2;  
    font-size: 12px;  
    margin-top: 40px;  
    padding: 50px 20px 20px;  
    ...
```

- ❖ We added the styles for our Application Decorators earlier.

SETTING THE DEFAULT APPLICATION DECORATOR

- ❖ A default application decorator also needs to be set in the `liferay-look-and-feel.xml`.

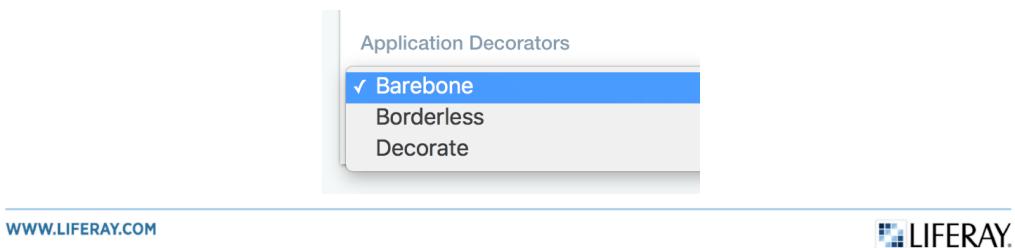
- ❖ In order to do this, we can simply use the line we see here:

```
<default-portlet-decorator>true</default-portlet-decorator>
```

- ❖ Once this is set, we'll be able to style the standard application decorator while being able to choose others when the need arises.

ADDING APPLICATION DECORATOR CSS

- The styling in the `_portlet_decorator.scss` is imported into the `_custom.scss`.
`@import "portlet_decorator";`
- Once your theme is deployed you can select your new application decorator.
- The new Trending option, for example, will be selectable from the *Application Decorators* drop-down list in the *Look-and-Feel Configuration* section of our applications.



THEME SETTINGS

- We can also add *Theme Settings* into our theme.
- *Theme Settings* are used to set either hard-coded or configurable values into the theme.
- These setting keys can be accessed inside FreeMarker theme files and, once deployed, are editable through the Control Panel.
- You can provide special settings in your theme that allow administrators to make stylistic changes from the platform.
- These settings provide stylistic flexibility within a theme.

EXERCISE: ADDING THEME SETTINGS

1. **Click** on the 02-theme-settings snippet.
2. **Copy** the contents of the snippet.
3. **Paste** the snippet contents over the `<!-- Insert snippet
02-theme-settings here -->` comment in the `liferay-look-and-feel.xml` file.
4. **Save** the file.

THEME SETTING PROPERTIES

- Each Theme Setting can have the following properties:
 - **configurable**: a value of true or false determines whether this field should be displayed or hidden from the Control Panel.
 - **key**: a name used to retrieve user value in the theme (no spaces)
 - **type**: defines the type of form field to show. Possible values are `text`, `textarea`, `select`, or `checkbox`
 - **options**: a comma separated list of options the user can choose for the `select` type.
 - **value**: default setting value

DEFAULT AVAILABLE SETTINGS

- ❖ There are some default Theme Settings that can be used in your theme without further configuration:
 - **Bullet Style:** Adds Bullet Point options for different applications
 - **Show Header Search:** Displays Header Search if set to true
 - **Show Maximize Minimize Application Links:** Gives a link to navigate between maximized and minimized view of an application
 - **Show Site Name Default:** Lets you control the Site name display
 - **Show Site Name Supported:** Lets you control the Site name support

```
<setting configurable="true" key="bullet-style" options="dots,arrows" type="select" value="dots" />
<setting configurable="true" key="show-header-search" type="checkbox" value="true" />
<setting configurable="true" key="show-maximize-minimize-application-links" type="checkbox" value="false" />
<setting configurable="false" key="show-site-name-default" value="true" />
<setting configurable="false" key="show-site-name-supported" value="true" />
```

USING CUSTOM THEME SETTINGS

- ❖ Custom Theme Settings that are added to the `liferay-look-and-feel.xml` can also be turned into variables.
- ❖ These variables are added in the `init_custom.ftl` file like so:

```
<#assign new_variable = getterUtil.getBoolean(themeDisplay.getThemeSetting("new-theme-setting")) />
```

- ❖ With the variable created, developers can add logic to their Theme Freemaker templates.
- ❖ Our theme has a new variable for the `show-header-search` Theme Setting in order to control how the search is displayed with this setting turned on inside the `navigation.ftl`.

CUSTOM SHOW HEADER SEARCH EXAMPLE

- ❖ The `show_header_search` variable was created in the `init_custom.ftl` using:
`<#assign show_header_search = getterUtil.getBoolean(themeDisplay.getThemeSetting("show-header-search")) />`
- ❖ In the `navigation.ftl`, this custom variable is used to control what displays:

```
<if show_header_search>
<div class="navbar-form navbar-right" role="search">
<div id="search">
...
</div>
<button aria-controls="navigation" class="btn-link btn-search hidden-xs"
type="button">
...
</button>
</div>
</if>
```

EXERCISE: LAYOUT TEMPLATE CONFIGURATION

- ❖ Earlier, we added the custom layout template source files into our `src/layouttpl` folder.
- ❖ We need to actually configure the `liferay-look-and-feel.xml` to include these layouts.

1. **Click** on the `03-layout-templates` snippet.
2. **Copy** the contents of the snippet.
3. **Paste** the snippet contents over the `<!-- Insert snippet 03-layout-templates -->` comment in the `liferay-look-and-feel.xml` file.
4. **Save** the file.

THE THEME PLUGIN PACKAGE PROPERTIES

- ❖ The `liferay-plugin-package.properties` file contains the default information for our theme.
- ❖ We'll revisit this file after we look at some other features of Liferay themes.

```
name=Space Program
module-group-id=liferay
module-incremental-version=1
module-version=1.0.0
tags=
short-description=
long-description=
change-log=
page-url=http://www.liferay.com
Provide-Capability=osgi.webresource;osgi.webresource=space-program-theme
author=Liferay, Inc.
licenses=LGPL
liferay-versions=7.0.0+
resources-importer-developer-mode-enabled=true
```

ADDING COLOR SCHEMES

- ❖ Developers can also include *Color Schemes* into the `liferay-look-and-feel.xml`.
- ❖ *Color Schemes* are variations of CSS and Images with a theme.
- ❖ Using this option, developers can effectively provide themes within themes for their administrators to configure based on the business needs.
- ❖ Let's look at how we would add these using the example of fire and ice Color Schemes.

COLOR SCHEMES IN THE XML

- ❖ Color Schemes can be added to the `liferay-look-and-feel.xml` by using the following:

```
<color-scheme id="01" name="Default">
  <default-cs>true</default-cs>
  <css-class>default</css-class>
  <color-scheme-images-path>${images-path}/color_schemes/${css-class}</color-scheme-images-path>
</color-scheme>
<color-scheme id="02" name="Fire">
  <css-class>fire</css-class>
</color-scheme>
```

- ❖ We need to set the default as well as any new Color Scheme we're adding.
- ❖ The `default-cs` makes sure to auto-select this color scheme.
- ❖ Each Color Scheme needs to include a `css-class` while the images are set using different variables in the default.

ADDING COLOR SCHEME CSS

- ❖ Once the XML is set, You can add the CSS changes using the following steps:

1. Create a new directory in `src/css` called `color_schemes`.
2. Add a new `scss` file for each Color Scheme in this folder.
 - Example: `_fire.scss` or `_ice.scss`
3. Add styling to these files using the class selector identified in the XML.

```
body.fire {
  color: #F00 !important;
}
```

4. Last, import the new `scss` files into the `_custom.scss`:

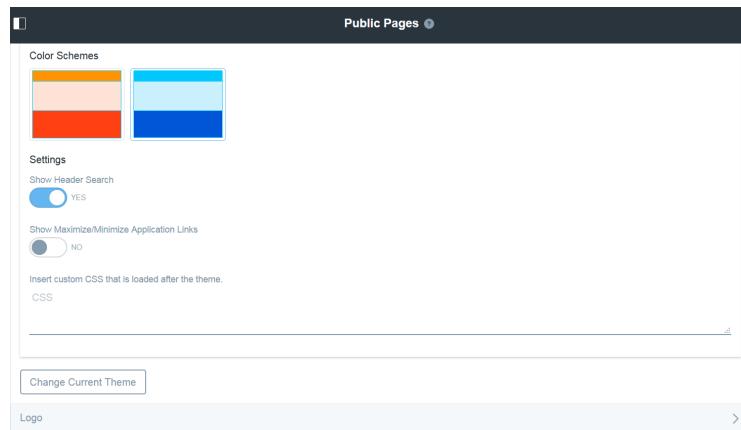
```
@import "color_schemes/fire";
@import "color_schemes/ice";
```

ADDING COLOR SCHEME IMAGES

- Next, images can be added using the following:
 1. Create a new directory in `src/images` called `color_schemes`.
 2. Add a new folders for each Color Scheme such as `color_schemes/fire` or `color_schemes/ice`.
 3. Place any images or thumbnails for the color schemes in their appropriate folder.
 - The thumbnail that will display in the page configuration menu needs to go here.

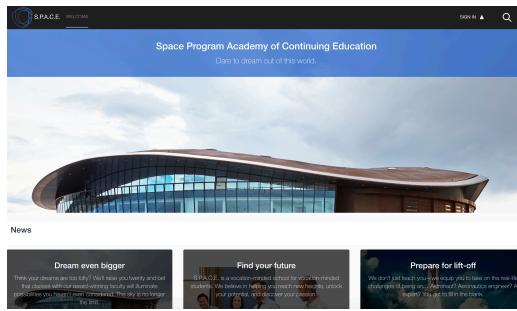
CONFIGURING PAGES

- Once Color Schemes and Theme Settings are added to the Theme, they can be accessed on each Site where the Theme is in use.



WHAT'S NEXT?

- ❖ This gives us a starting point for a working theme.
- ❖ Next, we'll look at some of the additional pieces we mentioned during our theme development.
- ❖ Items like Layout Templates deserve a closer look.
- ❖ We'll also take a look at additional tools and development options available.



WWW.LIFERAY.COM

 LIFERAY.

Notes:



THEMELETS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

S.P.A.C.E. SITE REQUIREMENTS

- So far, the Theme covers all of our global styles and responsive design.
- S.P.A.C.E. has some additional themes they want to create for their College and Department pages in the future.
- There are some styles and features they want to include in all of their future themes, including:
 - Making application moving on a page easier to visualize
 - Providing modern-style animation that emphasizes the menu when it's opened
- With this, they're looking for a way to provide a UI component that can be shared across themes and updated individually.
- We can take advantage of Liferay *Themelets* to accomplish this goal.

WHAT IS A THEMELET?

- ❖ Themelets are small, extendable, and reusable pieces of code that are implemented by a theme.
- ❖ They can consist of CSS, templates, images, and JavaScript just like a theme.
- ❖ They exist as *npm* packages and can be published to the *npm* registry for easy sharing and reuse.
- ❖ Themelets allow you to share pieces of code between themes, cutting down on boilerplate and repetition.
- ❖ You can either create new themelets or take advantage of existing themelets.

EXERCISE: VISUALIZING LAYOUT WITH THEMELETS

- ❖ The themelet generator is packaged within the `generator-liferay-theme` npm package.
1. **Run** `yo liferay-theme:themelet` in the command line or Terminal from your *liferay* folder.
 2. **Name** the themelet *Application Drag Indicator*.
 3. **Press** `Enter` to accept the default Id.
 4. **Choose** `7.0` for the version.
- ✓ Now we have a Themelet with an `src` folder and a `package.json`.

THEMELET STRUCTURE

- ❖ The `src` directory is where all themelet files exist that will eventually be used by a theme.
 - ❖ Note: A themelet can have multiple JS files. Any file with the extension `.js`, in `src/js` will automatically be injected into a theme during build.
- ❖ For files not CSS or JS, they will be made available in the built theme's files.

File	Description
<code>src</code>	Contains src files of themelet.
<code>src/css/_custom.scss</code>	Contains themelet styles that get automatically injected to theme on theme build.
<code>package.json</code>	Where themelet meta-data is defined.

EXERCISE: MAKING OUR THEMELET AVAILABLE

- ❖ In order to use your themelet in your theme, you must install it globally.
- ❖ There are two ways to make a themelet available to a theme:
 1. **npm link**: Creates a symlink in the global npm module directory
 2. **npm install -g**: Copies the files. If you use this method, you will have to re-run `npm install -g` to make any changes to your themelet available.
- ❖ To install your themelet as a global npm module, you can run either of these options from the root of the themelet.
 1. **Go to** the `application-drag-indicator-themelet` directory in the command line or the Terminal.
 2. **Run** the `npm link` command.
 - ❖ You will need admin access in the directory to run this command.
 - ❖ You may need to type `sudo npm link` on unix-based systems.
- ✓ Now we can add our dropzone code.

EXERCISE: ADDING APPLICATION DROPZONE CSS

- We can use this themelet to provide CSS that will allow us to visualize a page layout.
1. **Copy** the contents of the `application-drag-indicator.scss` file from `exercises/front-end-developer-exercises/03-theme-development/07-themelets`.
 2. **Paste** it into the `_custom.scss` file in your `application-drag-indicator-themelet/src/css`.
 3. **Save** the file.

EXERCISE: MENU ANIMATION THEMELET

- Now we can create the Menu Animation Themelet that we can reuse for any new theme.
1. **Run** `yo liferay-theme:themelet` in the command line or Terminal from your `liferay` folder.
 2. **Name** the themelet *Product Menu Animation*.
 3. **Press** `Enter` to accept the default Id.
 4. **Choose** `7.0` for the version.
- ✓ Now we have our new Themelet with an `src` folder and a `package.json`.

EXERCISE: TWO IS BETTER THAN ONE

- Just as before, we'll need to make this new Themelet available to our Theme.
 1. **Go to** the *product-menu-animation-themelet* directory in the command line or the Terminal.
 2. **Run** the `npm link` command.
 - You will need admin access in the directory to run this command.
 - You may need to type `sudo npm link` on unix-based systems.
- ✓ Now we can add the animation code!

EXERCISE: ADDING ANIMATION CSS

1. **Copy** the contents of the *product-menu-animation.scss* file from *exercises/front-end-developer-exercises/03-theme-development/07-themelets*.
 2. **Paste** it into the *_custom.scss* file in your *product-menu-animation-themelet/src/css*.
 3. **Save** the file.
- ✓ Now we can add our new themelets to the theme to deploy!

EXERCISE: INSTALLING OUR THEMELETS

1. **Open** the command line or Terminal.
2. **Go to** the root folder for the space-program-theme.
3. **Run** the *gulp extend* command.
 - There may be a brief pause while the command processes.
4. **Choose** *Themelet*.

EXERCISE: SELECTING BOTH THEMELETS

1. **Choose** *Search globally installed npm modules*.
 - Any globally-installed themelets should show up in the list.
2. **Press** Space to select the application-drag-indicator-themelet.
3. **Press** Space to select the product-menu-animation-themelet.
4. **Press** Enter to apply the themelet.
5. **Run** *gulp deploy* in your theme.
 - The themelet will now be added as an npm dependency to your theme.
 - Once deployed, this themelet will only display when changing where applications are placed on your page.

```
? Where would you like to search for themelets? Search globally installed npm modules (development purposes only)
? Select a themelet
➤ application-drag-indicator-themelet
@product-menu-animation-themelet
```

THEMELET CSS

- Themelet CSS and JS files will automatically be injected into your theme during the build task.
- Themelet CSS will be injected via Sass imports into the theme's `_custom.scss` file wherever the following CSS comment exists:

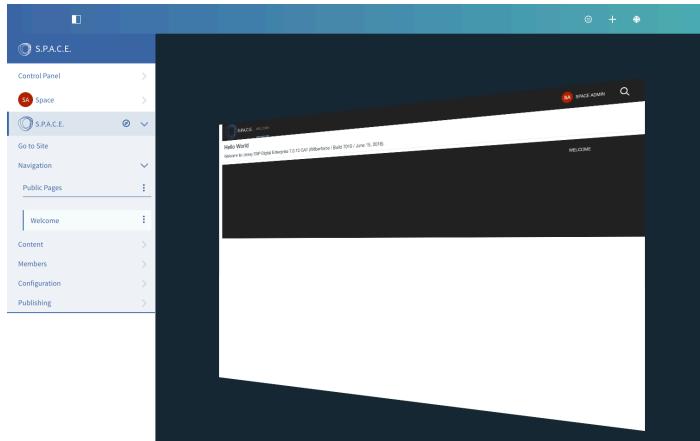
```
/* inject:imports */  
/* endinject */
```

THEMELET JAVASCRIPT

- Themelet JS will be injected via `<script>` tags into the theme's `portal_normal` template wherever the following HTML comment exists:
`<!-- inject:js -->
!-- endinject -->`
- These comments can be moved anywhere in the files they reside in. If you remove these comments, the themelet resources in question will not be added during the build.
 - *Note:* the `build` task is also run during the `deploy` and `deploy:gogo` tasks, so themelet files will also be injected during deploy.

SUCCESSFULLY ADDED

- Once deployed, we will see both the clear outline when moving applications around as well as the animation when we click on the Menu.



WWW.LIFERAY.COM

LIFERAY.

INSTALLING FROM THE NPM REGISTRY

- You can also choose from existing themelets on the npm registry.
- The list of themelets can be found here:
<https://www.npmjs.com/search?q=Themelet>
- Publishing to or extending from this list allows you to collaborate with other developers creating reusable pieces of code.

WWW.LIFERAY.COM

LIFERAY.

Notes:



THEME CONTRIBUTORS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

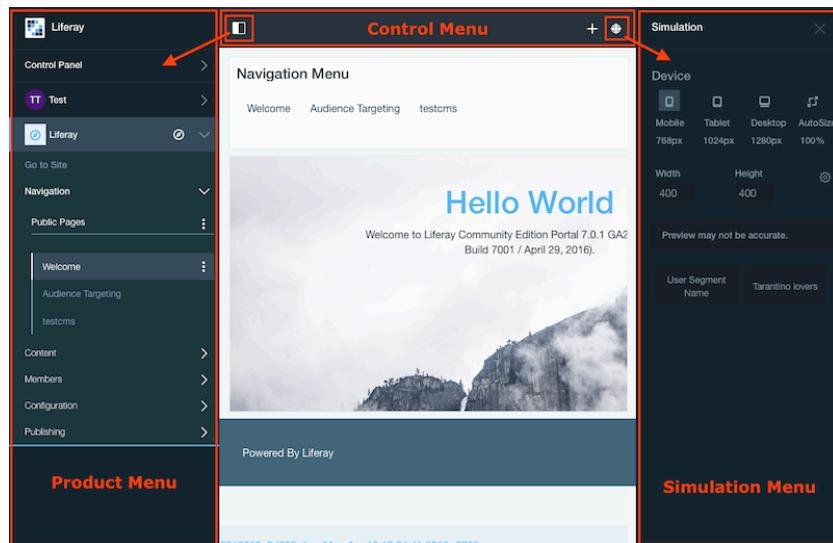
WHAT IS A THEME CONTRIBUTOR?

- ❖ A *Theme Contributor* is a module that contains UI resources independent of a theme.
- ❖ On deployment, the module is scanned for all valid CSS and JS files.
- ❖ These resources are then included on all pages, regardless of the current theme.
- ❖ If you'd like to package UI resources independent of a specific theme and include them on all pages, *Theme Contributors* are the right tool.

IDENTIFYING EXISTING THEME CONTRIBUTOR MODULES

- ❖ In previous Liferay versions, the standard UI for user menus and navigation, such as the Dockbar, was included in the theme template.
- ❖ Starting in Liferay DXP, these standard UI components are packaged as *Theme Contributors*.
- ❖ Specifically: the Control Menu, the Product Menu, and the Simulation Panel are now packaged as *Theme Contributor* modules.
- ❖ Styles for these UI components are handled outside of the theme.
- ❖ In order to edit the standard UI components, you'll need to create your own *Theme Contributor* to add your modifications on top of the existing components.
- ❖ You can also add new UI components to Liferay by creating new *Theme Contributor* modules.

MENU THEME CONTRIBUTORS



THE S.P.A.C.E. MENUS

- ❖ The S.P.A.C.E. front-end team may want to update the Control Panel, Product Menu, and Simulation Panel in order to make them feel like a part of unified experience with the new theme being created for the S.P.A.C.E. site.
- ❖ Using the power of *Theme Contributor* modules, the front-end team, with a little help from the back-end team, can create a new *Theme Contributor* to override the look and feel of the standard UI controls and update them to visually match their new theme.

CREATING YOUR OWN THEME CONTRIBUTOR

- ❖ To create a new *Theme Contributor*, you must first create an OSGi module.
 - You can use the Liferay *Blade CLI* tool to create the module.
- ❖ Next, you must identify your module as a *Theme Contributor* by adding the `Liferay-Theme-Contributor-Type` and `Web-ContextPath` properties to your `bnd.bnd` file.
 - `Liferay-Theme-Contributor-Type` helps Liferay identify your module as a *Theme Contributor*.
 - `Web-ContextPath` sets the context from which the *Theme Contributors* resources are hosted.

EXAMPLE BND.BND FILE

- ❖ Let's take a look at the Control Menu's bnd.bnd file.
- ❖ Notice the Liferay-Theme-Contributor-Type and Web-ContextPath headers in the file.

```
Bundle-Name: Liferay Product Navigation Control Menu Theme Contributor
Bundle-SymbolicName: com.liferay.product.navigation.control.menu.theme.contributor
Bundle-Version: 1.0.0
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Product Navigation
Liferay-Theme-Contributor-Type: product-navigation-control-menu
Web-ContextPath: /product-navigation-control-menu-theme-contributor
-include: ../../../../marketplace/web-content-management/bnd.bnd
```

MODIFYING STANDARD THEME CONTRIBUTORS

- ❖ If you want to edit or style standard *Theme Contributors*, you'll need to create a new, overriding *Theme Contributor*.
- ❖ In order to control *Theme Contributor* priority, you can set a *weight* for your *Theme Contributor* in your bnd.bnd file.
- ❖ You set the weight by adding the *Liferay-Theme-Contributor-Weight:* property to the *bnd.bnd* file.
Liferay-Theme-Contributor-Weight: 100
- ❖ *Theme Contributors* with higher *weight* will see their resources included later, thus overriding *Theme Contributors* with lower *weight*.

THEME CONTRIBUTOR RESOURCES

- ❖ Once the *weight* has been set, you can create a `src/main/resources/META-INF/resources` folder in your module and place your resources (CSS and JS files) in that folder.
- ❖ For example, in order to change the background color of the Control Menu, you could add `_control_menu.scss` to the `src/main/resources/META-INF/resources/css/blade.theme.contributor/` folder.
- ❖ In the `_control_menu.scss` file, you would add CSS to style the menu:

```
body {  
    .control-menu {  
        background-color: darkkhaki;  
    }  
}
```

IMPORTING RESOURCES

- ❖ All of the SCSS files are imported into the main `blade.theme.contributor.scss` file:

```
@import "bourbon";  
@import "mixins";  
  
@import "blade.theme.contributor/body";  
@import "blade.theme.contributor/control_menu";  
@import "blade.theme.contributor/product_menu";  
@import "blade.theme.contributor/simulation_panel";
```
- ❖ If you add your own SCSS files, remember to add them to the list of imports in the `blade.theme.contributor.scss` file.

WRAP-UP

- ❖ Once you have everything styled to your liking, build and deploy your new *Theme Contributor* module to see your modifications applied to pages and themes.
- ❖ Make sure to use *Theme Contributors* wisely.
- ❖ The UI contributions will affect every page and will remain, regardless of the theme.

Notes:



IMPORTING RESOURCES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

IMPORTING PREDEFINED CONTENT

- ❖ The Resources Importer allows you to deploy your themes with predefined content.
- ❖ This can be useful for creating content and themes together to provide a wholesale style change.
- ❖ When deployed, the Resources Importer creates a site template, which can be used for creating new sites with a predefined look and feel.

RESOURCES IMPORTER'S FILE STRUCTURE

- ❖ Theme resources reside in the source of your theme.
- ❖ You can create files and folders in a new *[theme-folder]/docroot/WEB-INF/src/resources-importer* directory.
- ❖ Example file structure:

```
resources-importer/
    document_library/
        documents/
    journal/
        articles/
        structures/
        templates/
    assets.json
    sitemap.json
```

RESOURCES IMPORTER CREATES A SITE TEMPLATE

- ❖ The site template that will be generated is defined in the *sitemap.json* file.
- ❖ This file describes the contents and hierarchy of a site that Liferay can import as a site template.
- ❖ The *sitemap.json* file defines the following:
 - Pages of a to-be-generated site template
 - Layout templates
 - Applications
 - Application preferences (Portlet preferences)
 - Content to display

RESOURCES IMPORTER EXAMPLE

- Here is a `sitemap.json` example that will create one page named `Welcome` that has two columns, a `Login` application in one and a `Hello World` application in the other.

```
{  
    "layoutTemplateId": "2_columns_ii",  
    "publicPages": [  
        {  
            "columns": [  
                [ { "portletId": "com_liferay_login_web_portlet_LoginPortlet"  
                [ { "portletId": "com_liferay_hello_world_web_portlet_  
                    HelloWorldPortlet" } ]  
            ],  
            "friendlyURL": "/home",  
            "name": "Welcome",  
            "title": "Welcome"  
        }  
    ]  
}
```

IMPORTING ASSETS WITH METADATA

- The `assets.json` file specifies details about the assets to be imported.
- Tags can be applied to any asset.
- Abstract summaries and small images can be applied to web content articles. As an example:

```
{  
    "assets": [  
        {  
            "name": "company_logo.png",  
            "tags": [ "logo", "company" ]  
        },  
        {  
            "abstractSummary": "This is an abstract summary.",  
            "name": "My Example.xml",  
            "smallImage": "company_logo.png",  
            "tags": [ "web content" ]  
        }  
    ]  
}
```

IMPORTING DOCUMENTS AND MEDIA

- ❖ By default, all assets under the directory `document_library/documents/` get imported into the platform's global *Documents and Media*.
- ❖ Example file structure:

```
document_library/
  documents/
    image.png
    Custom Folder/
      image 2.png
```
- ❖ With this example file structure, `image.png` will be placed in the root folder of the *Documents and Media* application.
- ❖ `image 2.png` will be placed in a folder named `Custom Folder`.

IMPORTING WEB CONTENT

- ❖ The journal directory is used for importing various assets related to web content such as structures (JSON), templates (Velocity/FreeMarker), and web content articles (XML).

```
journal/
  articles/
    My Example Template A/ (matches Template name)
      My Example Article X.xml
      My Example Article Y.xml
  structures/
    My Example Structure H.json
  templates/
    My Example Structure H/ (matches Structure name)
      My Example Template A.fltr
      My Example Template B.fltr
```

EXAMPLE STRUCTURE

- Here is an example of a Structure. We'll call it My Example Structure H.json:

```
{  
    "availableLanguageIds": [ "en_US" ],  
    "defaultLanguageId": "en_US",  
    "fields": [  
        { "name": "Header", "type": "text", ... },  
        { "name": "Body", "type": "textarea", ... }  
    ]  
}
```

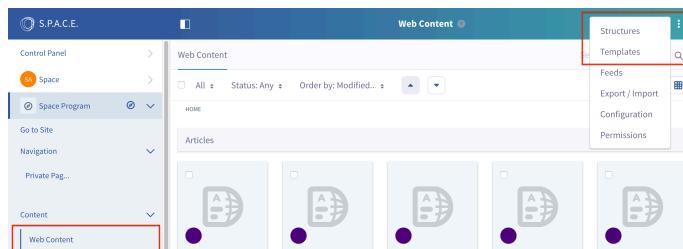
- The source JSON of Web Content Structures can be found by clicking the Source tab while editing the Structure.

EXAMPLE TEMPLATE

- Here's an example of a Template. We'll call it My Example Template A.ftl:

```
<h1>${Header.getData()}</h1>  
  
<p>${Body.getData()}</p>
```

- Both Structures and Templates can be created/edited by navigating to Web Content in Site Administration via the Product Menu.



EXAMPLE ARTICLE X.XML

- ❖ To access the source XML of an article, go to the edit page of the article and click the *View Source* option.
- ❖ Let's take a look at an example article. We'll call it Example Article X.xml:

```
<?xml version="1.0"?>

<root available-locales="en_US" default-locale="en_US">
    <dynamic-element name="Header" type="text" index-type="keyword"
        instance-id="mdyl">
        <dynamic-content language-id="en_US"><! [CDATA[My Header]]>
        </dynamic-content>
    </dynamic-element>
    <dynamic-element name="Body" type="text_box" index-type="keyword"
        instance-id="opiq">
        <dynamic-content language-id="en_US"><! [CDATA[My body <em>with</em>
        HTML.]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

ADDING WEB CONTENT TO THE SITEMAP

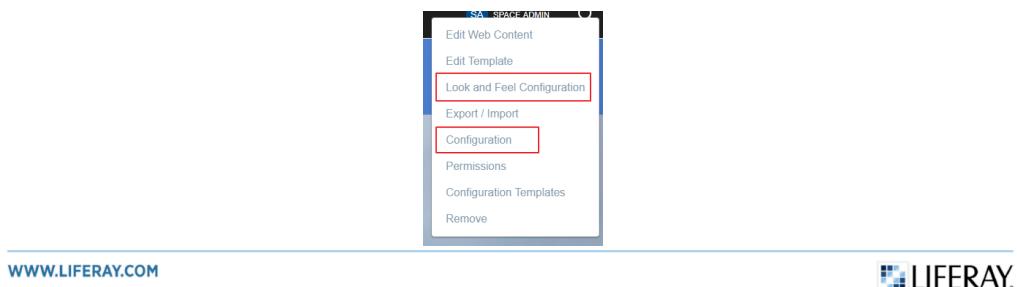
- ❖ To add a Web Content Display and choose a content article to display, we could do something like this example in the sitemap.json:

```
"columns": [
    [
        {
            "portletId": "com_liferay_hello_world_web_portlet_HelloWorldPortlet"
        },
        {
            "portletId":
                "com_liferay_journal_content_web_portlet_JournalContentPortlet",
            "portletPreferences": {
                "articleId": "My Example Article X.xml",
                "groupId": "${groupId}",
                ...
            }
        },
        ...
    ],
    ...
]
```

- ❖ Here, we'll add the web content application right below the Hello World application.

PORLET PREFERENCES

- ❖ You'll notice we use a `portletPreferences` property to select the article we will display in the Web Content Display application.
- ❖ `portletPreferences` are properties that store basic application configuration data.
- ❖ Just as you can apply some configurations for applications through the UI, `portletPreferences` properties allow you to set configurations for applications in the `sitemap.json` file.



COMMON PORTLET PREFERENCES

- ❖ Here are a few commonly used `portletPreferences` properties:

<code>articleId</code>	Selects an article
<code>groupId</code>	Selects a site
<code>portletSetupTitle_en_US</code>	Sets custom title
<code>portletSetupUseCustomTitle</code>	Allows use of custom title
<code>portletSetupPortletDecoratorId</code>	Sets Application Decorator

- ❖ Some `portletPreferences` properties are general, while others are application-specific.

EXAMPLE PORTLET PREFERENCES

- In the previous example, we are taking advantage of `portletPreferences` to configure the Web Content Display application.

```
"portletPreferences": {  
    "articleId": "My Example Article X.xml",  
    "groupId": "${groupId}",  
    ...  
}
```

- With the `articleId` property, we choose which article to display in the Web Content Display application.
 - The Example Article X.xml will be located in the journal/articles folder of our example Resource Importer module.
- The `groupId` property determines the site where the content will be displayed.

SETTING APPLICATION DECORATORS

- It is also possible to set the application decorator from a `sitemap.json` where `portletSetupPortletDecoratorId` is the id of the decorator to be used:

```
{  
    "portletId": "com_liferay_journal_content_web_portlet_  
    JournalContentPortlet",  
    "portletPreferences": {  
        "articleId": "My Content.xml",  
        "groupId": "${groupId}",  
        "portletSetupPortletDecoratorId": "barebone"  
    }  
}
```

- In this example, the application decorator is set to the `barebone` setting.
- The `portletSetupPortletDecoratorId` property can be set to any of the application decorators available (i.e., `barebone`, `borderless`, `decorate`).

SETTING CUSTOM DECORATORS

- ❖ The `portletSetupPortletDecoratorId` preference could be set to our custom decorator id as well. The preference can be set when embedding applications in a theme:

```
<#assign VOID = freeMarkerPortletPreferences.setValue  
("portletSetupPortletDecoratorId", "barebone") />  
<div aria-expanded="false" class="collapse navbar-collapse"  
id="navigationCollapse">  
    <if has_navigation && is_setup_complete>  
        <nav class="${nav_css_class} site-navigation" id="navigation"  
        role="navigation">  
            ...  
        </nav>  
    </if>  
</div>  
  
<#assign VOID = freeMarkerPortletPreferences.reset() />
```

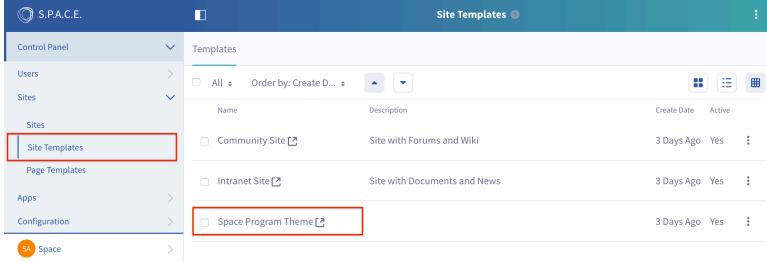
- ❖ Application decorators can also be updated from the application's *Look and Feel* menu.

EXERCISE: BUNDLING RESOURCES IN OUR THEME

1. **Go to** `space-program-theme/src/WEB-INF`.
2. **Create** a folder named `src`.
3. **Copy** the `resources-importer` folder from
`exercises/front-end-developer-exercises`
`/03-theme-development/09-importing-resources`.
4. **Paste** the entire folder into
`space-program-theme/src/WEB-INF/src`.

DEPLOYING RESOURCES

- With the necessary `sitemap.json` and resources, you can deploy the theme.
- With the default configuration, the Resources Importer will create a Site Template that shares the name of the theme.
- We can also create a new site that includes the resources on the page.
- We'll do this to create a separate Space Program site that will display different assets from S.P.A.C.E.



The screenshot shows the S.P.A.C.E. Control Panel interface. On the left, there's a sidebar with 'Control Panel' at the top, followed by 'Users', 'Sites' (which is expanded), 'Page Templates', 'Apps', 'Configuration', and 'Space' (which is also expanded). Under 'Space', 'Site Templates' is highlighted with a red box. The main area is titled 'Site Templates' and shows a table with three rows. The first row is 'Community Site' (Site with Forums and Wiki), the second is 'Intranet Site' (Site with Documents and News), and the third is 'Space Program Theme' (highlighted with a red box). Columns include 'Name', 'Description', 'Create Date', and 'Active'.

IMPORTING RESOURCES TO EXISTING SITES

- To configure the Resources Importer to import resources into a site, rather than a site template, add the following properties to:
`{theme-name}/src/WEB-INF/liferay-plugin-package.properties`.
...
`resources-importer-target-class-name=com.liferay.portal.kernel.model.Group`
`resources-importer-target-value=[site-name]`
...
- If the `site-name` is an existing site, the Resources Importer will import into that site.
- When the site doesn't exist, the Importer will create it for you.
 - **Note:** This can be a great way to show off your theme's features, or demonstrate how to arrange content nicely.

EXERCISE: UPDATING OUR PACKAGE PROPERTIES

1. **Go to** the exercises/front-end-developer-exercises/03-theme-development/09-importing-resources folder.
2. **Copy** the liferay-plugin-package.properties file.
3. **Go to** the space-program-theme/src/WEB-INF.
4. **Paste** the file to overwrite the existing liferay-plugin-package.properties.
5. **Run** gulp deploy.

SELECTING THE THEME

- The Resources Importer created a new site called *Space Program*, but the theme may not be automatically applied to it.
- You may need to manually apply the theme to the *Space Program* site to see the full styling:
 1. **Click** the Site Selector under Site Administration in the Menu.
 2. **Click** the My Sites tab.
 3. **Choose** the *Space Program* site.
 4. **Click** Private Pages under Navigation.
 5. **Click** Options→Configure to the right of *Private Pages*.
 - **Note:** if the Options menu is not visible, you may need to use the Options menu on *Private Pages*.
 6. **Click** Change Current Theme.
 7. **Choose** the *Space Program* theme.
 8. **Click** Save.

DEVELOPER MODE

- The Resources Importer has a developer mode, which deletes and re-creates the target site or site template on each deploy.
- To enable developer mode, add the following to:
`liferay-plugin-package.properties`
...
`resources-importer-developer-mode-enabled=true`
...
- Themes created via the Liferay Theme Generator have developer mode enabled by default.
- Although this is useful for development, it should never be used in a production environment.

Notes:



EMBEDDING APPLICATIONS INTO THEMES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

EMBEDDING APPLICATIONS

- ❖ It's often necessary for developers to embed applications into their themes.
- ❖ Embedding applications into the `portal_normal.ftl` will render them on each site page.
- ❖ Let's look at a couple of ways we can embed applications.

TAGLIBS USED IN EMBEDDING APPLICATIONS

- ❖ There are three taglibs that we can use in our theme to embed applications or content:
 - <@liferay_portlet["runtime"]
 - <@liferay_journal["journal-article"]
 - <@liferay_ui["asset-display"]
- ❖ Each of these taglibs takes different parameters.

USING PORTLET PROVIDER CLASS NAME

- ❖ The <@liferay_portlet["runtime"] expects two parameters:
 1. portletProviderAction requests the portlet provider to perform an action for display.
 - Using portletProviderAction.VIEW for the first parameter most commonly used displays the default application view.
 2. The portletProviderClassName requires the fully qualified class name of the entity on which we want to perform the action.
- ❖ The portletProviderClassName is always coupled with the portletProviderAction.

```
<@liferay_portlet["runtime"]  
    portletProviderAction=portletProviderAction.VIEW  
    portletProviderClassName="CLASS.NAME"  
/>>
```

APPLICATIONS IN THE TAGLIB

- ❖ Using the above method will work for a set of applications that can be found in the source code.
- ❖ To find which applications work, follow these steps:
 1. You can go to <https://github.com/liferay/liferay-portal>
 2. Search the code for *extends BasePortletProvider*
- ❖ From there, you will be able to find the list of applications and what actions you can use with them.

EMBEDDING WEB CONTENT USING LIFERAY PORTLET TAGLIB

- ❖ For other applications, such as Web Content, you would need to pass in the *portletName*.
- ❖ Here is an example of adding Web Content using

```
<@liferay_portlet["runtime"]>  
<#assign VOID = freeMarkerPortletPreferences.setValue  
("portletSetupPortletDecoratorId", "barebone") />  
<#assign VOID = freeMarkerPortletPreferences.setValue  
("groupId", "${group_id}") />  
<#assign VOID = freeMarkerPortletPreferences.setValue  
("articleId", "ARTICLE_ID") />  
  
<@liferay_portlet["runtime"]  
defaultPreferences ="${freeMarkerPortletPreferences}"  
portletProviderAction=portletProviderAction.VIEW  
instanceId="INSTANCE_ID"  
portletName="com_liferay_journal_content_web_portlet_JournalContentPortlet"  
/>  
  
<#assign VOID = freeMarkerPortletPreferences.reset() />
```

THE PORTLET NAME ATTRIBUTE

- ❖ The portletName is the application id, written as the string reference of the application class path.

```
<@liferay_portlet["runtime"]  
    portletName="CLASS_NAME"  
/>
```

- ❖ For example, the Web Content application would be
`com_liferay_journal_content_web_portlet
_JournalContentPortlet.`

PORTLET PREFERENCES FOR EMBEDDED APPLICATIONS

- ❖ It is also possible to set preferences in an application using `#{freeMarkerPortletPreferences}`, as we can see in the Web Content example.
- ❖ This allows you to change the application preferences and have it immediately display in the theme.

```
<#assign VOID = freeMarkerPortletPreferences.setValue(  
    "portletSetupPortletDecoratorId", "barebone") />  
  
<@liferay_portlet["runtime"]  
    defaultPreferences="#{freeMarkerPortletPreferences}"  
    portletName="com_liferay_login_web_portlet_LoginPortlet"  
/>  
  
<#assign VOID = freeMarkerPortletPreferences.reset() />
```

PORLET RUNTIME ATTRIBUTES

- Let's look at some of the additional attributes that can be used:
 - **defaultPreferences**: This is a string of Portlet Preferences for the application that will be rendered. It could include look and feel configurations.
 - **instanceId**: If the application is instanceable, this allows for the instance id to be set.
 - **persistSettings**: This attribute will have an application use its default settings, which will persist across layouts. By default the attribute is set to *true*.
 - **settingsScope**: This attribute specifies which settings the application is to use. The default setting is *portletInstance* but can be set to *group* or *company*.

USING THE JOURNAL ARTICLE TAGLIB

- You can also embed Web Content using the `<@liferay_journal["journal-article"]>` taglib.
`<@liferay_journal["journal-article"]>`
 `articleId="ARTICLE_ID"`
 `ddmTemplateKey="TEMPLATE_KEY"`
 `groupId=${group_id}`
 `/>`
- The `<@liferay_journal["journal-article"]>` taglib requires the following:
 - **Article ID**: The id of the Web Content Article you wish to display
 - **Template Key**: The id of any Web Content Template you want to identify
 - **groupId**: The Site id where the content is available

USING THE ASSET DISPLAY TAGLIB

- Finally, you can also embed other specific assets, such as wiki articles or blogs, using the `<@liferay_ui["asset-display"]` taglib.

```
<@liferay_ui["asset-display"]
  className="JAVA_CLASS_NAME"
  classPK="CLASS PK (RESOURCE PK) OF ASSET"
  template="full_content"
/>>
```

- The `<@liferay_ui["asset-display"]` taglib requires the following:
 - **Class Name:** The Java Class Name of the asset
 - This would be the content type, such as blogs or documents
 - **Class PK:** The Primary Key id of the specific asset to display
 - This would be the specific blog or document you want to display
 - **Template:** This identifies the template used to display the asset

EMBEDDING WITH THE RIGHT TAGLIB

- In the past, the only option was to embed applications themselves.
- With these taglibs, you can choose to embed applications, specific web content, or any other asset you'd like on display.
- This gives you the flexibility to choose the option that best fits your requirements.

Notes:

Chapter 4

Layout Templates



CONTROLLING PAGE LAYOUTS

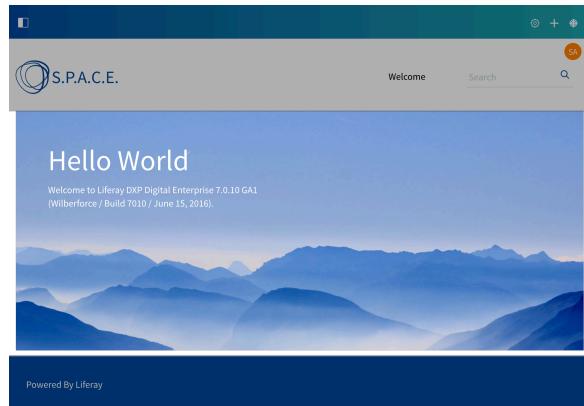
Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

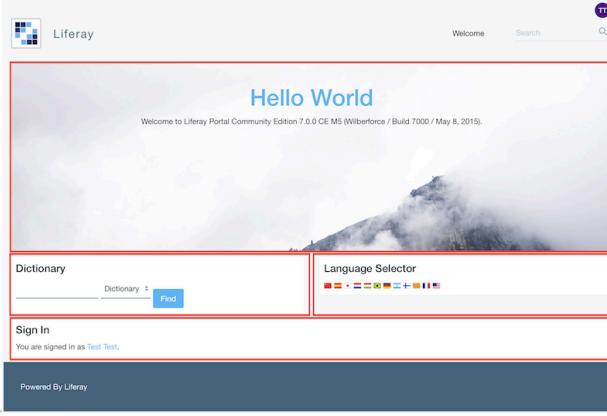
EXAMINING THE LIFERAY PAGE

- ❖ A Site page has three main sections.
 1. The Header
 2. The Content Section
 3. The Footer
- ❖ As we know, the Theme is responsible for providing the global styling, as well as header and footer styling.
- ❖ *Layout Templates* control the positioning of the content between the header and the footer.



CONTROLLING DISPLAY

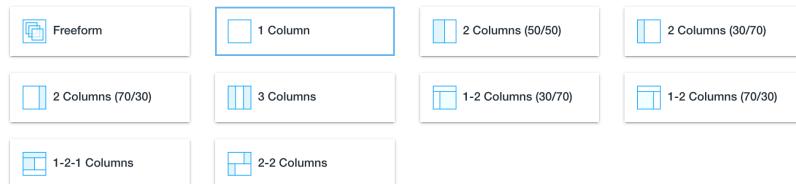
- ❖ Multiple applications can be added to the page that align with the Layout Template.
- ❖ Applications stack vertically and fill the available width of its parent column.



The screenshot shows a Liferay portal page with a red border around the main content area. Inside, there's a 'Hello World' header, a 'Dictionary' application, a 'Language Selector' application, and a 'Sign In' application. The 'Dictionary' and 'Sign In' boxes are explicitly outlined with red borders. At the bottom, there's a dark footer bar with the text 'Powered By Liferay'. Below the screenshot, the URL 'WWW.LIFERAY.COM' is displayed, along with the Liferay logo.

PREDEFINED LAYOUT TEMPLATES

- ❖ Liferay comes bundled with a number of predefined layout templates you can choose from.



- ❖ If none of the provided layout templates suit your needs, custom layout templates can be created and deployed to accomplish any combination of rows/columns.

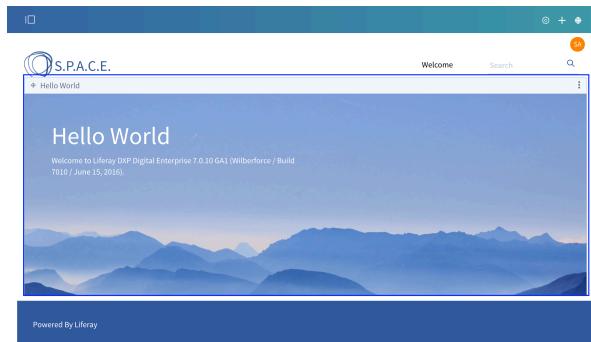
ROWS AND COLUMNS

- ❖ Each layout can have any number of rows with columns.
- ❖ Columns in a layout use Bootstrap's Grid system (see: <http://getbootstrap.com/css/#grid>) to determine screen and column size.
- ❖ There are 12 sections total in the fluid grid system, and you can divide them up any way you like.
- ❖ Because it's part of Bootstrap's grid system, we never have to specify percentages or pixel widths, and the layouts are also responsive.



THE 1 COLUMN LAYOUT TEMPLATE

- ❖ We can see the *1 Column* layout template on the Welcome page of S.P.A.C.E.
- ❖ Right now, the *Hello World* application is contained in the single column.
- ❖ Applications on this page will stack either above or below the *Hello World Application*.

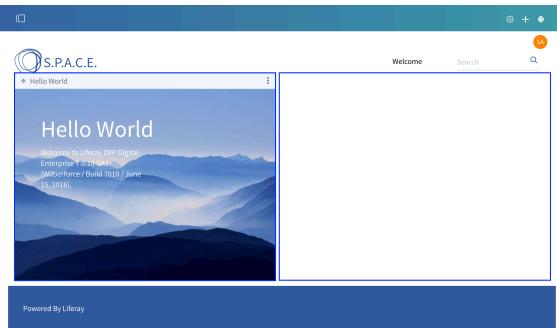


THE 2 COLUMN 50/50 LAYOUT TEMPLATE

- ❖ As another example, the *2 Columns (50/50)* layout has two size 6 columns, side by side.



- ❖ If we changed our layout to this one, we could place two applications side by side.

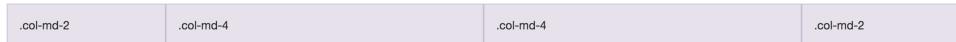


WWW.LIFERAY.COM

LIFERAY.

THE S.P.A.C.E. CUSTOM LAYOUT

- ❖ S.P.A.C.E. would like a custom layout that can include four applications in one row, with the primary page real estate given to the middle applications.
- ❖ This layout can be used to display the main content in the middle, with some static content for campaigns or advertisements on the sides.
- ❖ Next, we'll walk through creating a custom layout template to accomplish these goals.



WWW.LIFERAY.COM

LIFERAY.

Notes:



CREATING LAYOUT TEMPLATES WITH THE LIFERAY THEME GENERATOR

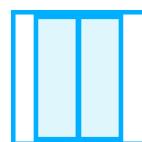
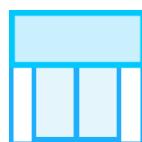
Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

S.P.A.C.E. LAYOUT REQUIREMENTS

- ❖ Part of the modern design S.P.A.C.E. is looking for on their website includes how applications are positioned on the page.
- ❖ Pages may require different layouts depending on where they are in the site.
- ❖ The construction of our S.P.A.C.E. sites and pages may be the responsibility of our Administration and Content teams, but we can provide the layout design needed.
- ❖ In addition to the basic layouts, we need a couple of layouts that conform to the following models for our top and secondary level pages:



USING THE LIFERAY THEME GENERATOR FOR LAYOUTS

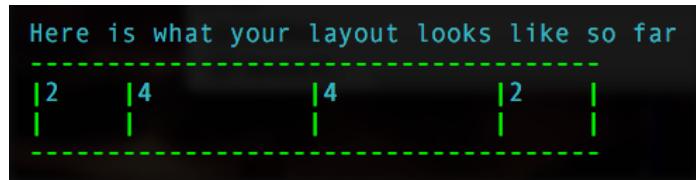
- We have seen how the Liferay Theme Generator can be used to create themes and themelets.
- We can also use the generator to create layout templates by using:
`yo liferay-theme:layout`
- This will create a new folder for your layout template in your current working directory.
- Let's walk through creating one of the layout templates from our theme.

EXERCISE: CREATING A LAYOUT TEMPLATE

1. **Go to** the *liferay* folder you created in the command line or Terminal.
 - Windows: *C:\liferay*
 - Mac/Linux: *~/liferay*
2. **Run** the `yo liferay-theme:layout` command.
3. **Type** the name *Space 50/50 Width Limited*.
4. **Type** *space-50-50-width-limited* for the template id.
5. **Choose** 7.0 for the version.
 - At this point, the layout template design process begins.
 - As the generator states, layout templates implement Bootstrap's grid system.
 - Every row consists of 12 sections, so columns range in size from 1 to 12.
 - The sub-generator is user-friendly, allowing you to add and remove rows and columns as you design.

EXERCISE: MODIFYING ROWS AND COLUMNS

1. **Type** 4 when the sub-generator prompts you for a number of columns.
 2. **Press Enter.**
 3. **Choose** the following for each column:
 - **Column 1:** 2/12 - 16.66%
 - **Column 2:** 4/12 - 33.33%
 - **Column 3:** 4/12 - 33.33%
 - **Column 4:** 2/12 - 16.66%
- ✓ This will show us how it looks so far!



EXERCISE: FINISHING OUR FIRST SPACE TEMPLATE

- After we've created the first row and columns, you will see the following options:
 - **Add row:** adds a row below the last row created
 - **Insert row:** allows you to inject a new row somewhere in the existing layout
 - **Remove row:** allows you to remove a row from the current layout
 - **Finish row:** generates layout template files
1. **Choose Finish Layout.**
 2. **Type** in the command line or Terminal:
 - Windows: `C:\liferay\bundles\liferay-dxp-digital-enterprise-[version]\tomcat-[version]`
 - Mac/Unix: `~/liferay/bundles/liferay-dxp-digital-enterprise-[version]/tomcat-[version]`
 3. **Press Enter** to choose the default `http://localhost:8080`.

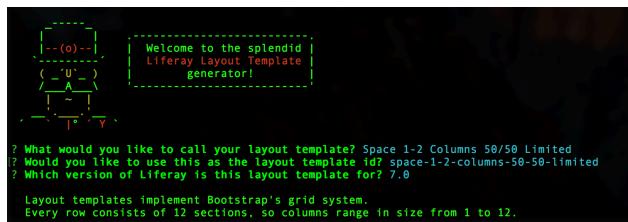
PACKAGING LAYOUTS IN THEMES

- As we saw with our Space Program Theme, we can bundle content and layouts with themes.
- This layout has already been included in the theme using the following steps:
 - In [your-theme]/src, create a layouttpl folder.
 - Copy and paste the tpl and png file from [your-layout]/docroot to [your-theme]/src/layouttpl.
 - Last, you can copy the <layout-template> structure from the liferay-layout-templates.xml found in WEB-INF.
 - Paste it into the theme's liferay-look-and-feel.xml.
- But we can use *gulp* commands to deploy layouts separately as well.

EXERCISE: CREATING THE 2 ROW LAYOUT

- Let's create our 2 row layout that uses a similar design, but includes a row at the top for banner images and carousels.
- We can deploy this one individually to the platform.

1. **Run** the *yo liferay-theme:layout* command from the *liferay* folder.
2. **Type** the name *Space 1-2 Columns 50/50 Limited*.
3. **Type** *space-1-2-columns-50-50-limited* for the template id.
4. **Choose** *7.0* for the version.



EXERCISE: ADDING THE TOP ROW

1. **Type 1** when the sub-generator prompts you for a number of columns.
 2. **Press Enter.**
 3. **Press Enter** when you see the following:

12/12 - 100% - | - only available width, hit enter

✓ This will show us how it looks so far!

```
? How many columns would you like for row 1? 1
? How wide do you want row 1 column 1? 12/12 - 100% - | [ ]
```

Here is what your layout looks like so far

```
-----  
|12|
```

EXERCISE: FINISHING THE SECOND ROW

1. **Press** Enter next to Add row.
 2. **Type** 4 when the sub-generator prompts you for a number of columns.
 3. **Press** Enter.
 4. **Choose** the following for each column:
 - ▶ Column 1: $2/12 - 16.66\%$
 - ▶ Column 2: $4/12 - 33.33\%$
 - ▶ Column 3: $4/12 - 33.33\%$
 - ▶ Column 4: $2/12 - 16.66\%$

✓ Now we can see how the rows look together.

```
Here is what your layout looks like so far  
-----  
| 12 |  
|  
|-----  
| 2 | 4 | 4 | 2 |
```

EXERCISE: FINISHING OUR SECOND TEMPLATE

1. **Choose** *Finish Layout*.
2. **Type** in the command line or Terminal:
 - Windows: `C:\liferay\bundles\liferay-dxp-digital-enterprise-[version]\tomcat-[version]`
 - Mac/Unix: `~/liferay/bundles/liferay-dxp-digital-enterprise-[version]/tomcat-[version]`
3. **Press** *Enter* to choose the default `http://localhost:8080`.

EXERCISE: ADDING A NEW THUMBNAIL

- Now that the layout is created, we want to add our thumbnail to replace the default.
 - This will make it clear, visually, which layout to select on the platform.
1. **Go to** `exercises/front-end-developer-exercises /o4-layout-templates/`.
 2. **Copy** the `space_1_2_columns_50_50_limited.png` file.
 3. **Go to** your `space-1-2-columns-50-50-limited-layouttpl/docroot` folder found in your `liferay` folder.
 4. **Replace** the default png with the png you just copied.
- ✓ Now we're ready to deploy!

EXERCISE: DEPLOYING THE NEW LAYOUT TEMPLATE

1. **Go to** the base layout folder in the command line or Terminal.
 - **Windows:** `C:\liferay\space-1-2-columns-50-50-limited-layouttpl`
 - **Mac/Linux:** `~/liferay/space-1-2-columns-50-50-limited-layouttpl`
 2. **Run** the `gulp deploy` command.
- ✓ Now our new layout is available on the platform!

The screenshot shows the Liferay Themes Generator's layout configuration screen. At the top, there's a dropdown menu labeled 'Type' with 'Layout' selected. Below it, a note says 'Create an empty page you can lay out manually.' A grid of layout options is shown in three rows. The first row contains 'Freeform', '1 Column', '2 Columns (50/50)', and '2 Columns (30/70)'. The second row contains '2 Columns (70/30)', '3 Columns', '1-2 Columns (30/70)', and '1-2 Columns (70/30)'. The third row contains '1-2-1 Columns', '2-2 Columns', and 'Space 1-2 Columns 50/50 ...', with the last one being the active selection, indicated by a blue border.

LAYING IT OUT

- The Themes Generator makes new layouts quick and easy!
- After creating a new layout using the generator, you will see `docroot/{layout-name}.tpl` in your layout template directory that reflects the layout you created.
- The generator will also install the `npm` dependencies used for deployment.
- After running `gulp deploy`, you will also see a `dist` folder containing the application file itself.
- This file can then be deployed on any server you're working with.

Notes:



TYPICAL ELEMENTS AND CLASSES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

PYGON LAYOUT TEMPLATE SOURCE

- ❖ Let's take a look at the PYGON Layout source code.

```
<div class="porygon-50-50-width-limited" id="main-content" role="main">
  <div class="portlet-layout row">
    <div class="col-md-2 portlet-column portlet-column-first" id="column-1">
      $processor.processColumn("column-1", "portlet-column-content"
        portlet-column-content-first")
    </div>
    <div class="col-md-4 portlet-column" id="column-2">
      $processor.processColumn("column-2", "portlet-column-content")
    </div>
    <div class="col-md-4 portlet-column" id="column-3">
      $processor.processColumn("column-3", "portlet-column-content")
    </div>
    <div class="col-md-2 portlet-column portlet-column-last" id="column-4">
      $processor.processColumn("column-4", "portlet-column-content"
        portlet-column-content-last")
    </div>
  </div>
</div>
```

COLUMN ELEMENTS AND CLASSES

- Let's take a closer look at the first column of the previous example and the various elements it consists of.

```
<div class="col-md-2 portlet-column portlet-column-first" id="column-1">  
    $processor.processColumn("column-1", "portlet-column-content  
    portlet-column-content-first")  
</div>
```

- You'll notice three things that we'll work through:
 1. The id
 2. The classes
 3. the `$processor.processColumn`

COLUMN CONTAINER

- **The id: column-1**
 - Unique identifier for the column that matches ID passed to `$processor.processColumn`.
- **Classes: col-md-2**
 - This class comes from Bootstrap's grid system and determines two things: the percentage-based width of the element, and the media query breakpoint for when this element expands to 100% width.
 - 12 is the maximum amount, so `col-md-2` indicates 2/12 width, or 16.66%.
- **Classes portlet-column portlet-column-first**
 - All column containers should have the `portlet-column` class.
 - For rows with more than one column, the first column will have `portlet-column-first` and the last will have `portlet-column-last`.
 - For rows with only one column, the column will have the `portlet-column-only` class.

INSIDE \$PROCESSOR.PROCESSCOLUMN

- `$processor.processColumn` is necessary to render our layouts.
- The `$processor.processColumn` function takes the CSS column ID and the list of CSS classes as its two arguments.
 - `$processor.processColumn: column-1`
 - Unique identifier; should match ID of the parent div
 - `$processor.processColumn: portlet-column-content`
`portlet-column-content-first`
 - Additional classes added to the content element; classes match the parent div's classes with `-content` appended

MODIFYING TEMPLATE BREAKPOINTS

- When looking at the example template, you'll notice this Bootstrap grid class used on every column.
`col-md-{size}`
 - The different sizes available are `xs`, `sm`, `md`, and `lg`.
 - The medium size is used by default, but the others can be used in layout templates as well.
 - For example, setting the column classes to `col-lg-{size}` means the columns would expand to 100% width at a larger screen width than `col-md-{size}`.
 - These classes can also be mixed to achieve more advanced layouts.

BREAKPOINT EXAMPLE

- ❖ In this example row, on medium-sized view ports (such as a tablet), column-1 will be 33.33% width.
- ❖ column-2 will be 66.66% width.
- ❖ On small-sized view ports, both column-1 and column-2 will be 50% width.

```
<div class="portlet-layout row">
    <div class="col-md-4 col-sm-6 portlet-column portlet-column-first"
        id="column-1">
        $processor.processColumn("column-1", "portlet-column-content
        portlet-column-content-first")
    </div>
    <div class="col-md-8 col-sm-6 portlet-column portlet-column-last"
        id="column-2">
        $processor.processColumn("column-2", "portlet-column-content
        portlet-column-content-last")
    </div>
</div>
```

Notes:



EMBEDDING APPLICATIONS INTO LAYOUT TEMPLATES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

EMBEDDING APPLICATIONS

- It is possible to embed an application in your layout template.
- This is very similar to embedding an application into the theme.
- The major difference is that administrators can be more flexible with where to apply a layout.
- If the application does not have a Java class that extends `BasePortletProvider`, developers can use `$theme.runtime()`.
- If it does, this is done using
`$processor.processPortlet("CLASS_NAME", ACTION)`.

USING \$THEME.RUNTIME()

- ❖ In Liferay 6.2, you would pass the application id as a parameter into `runtime()`.
- ❖ In DXP, the application id is no longer a number but a snake case string of the application class path.
- ❖ For example, in 6.2, you could write `$theme.runtime('58')` to embed the login application into their layout.
- ❖ In DXP, you can accomplish the same thing by passing `$theme.runtime('com_liferay_login_web_portlet_LoginPortlet')`.

\$THEME.RUNTIME() EXAMPLE

- ❖ Here is an example where the Login application is embedded into a layout:

```
<div class="columns-1-2" id="main-content" role="main">
    <div class="portlet-layout">
        <div class="portlet-column portlet-column-only" id="column-1">
            $theme.runtime("com_liferay_login_web_portlet_LoginPortlet")

            $processor.processColumn("column-1", "portlet-column-content
            portlet-column-content-only")
        </div>
    </div>
    ...
</div>
```

\$PROCESSOR.PROCESSPORTLET("CLASS_NAME", ACTION)

- The \$processor.processPortlet declaration, just as the theme declaration, expects two parameters:
 - The class name of the entity type you want the application to handle
 - The type of action
- For example, if you wanted to embed the Breadcrumb application into the layout, you could use the following:

```
$processor.processPortlet("com.liferay.portal.kernel.  
servlet.taglib.ui.BreadcrumbEntry", $portletProviderAction.VIEW)
```

WHERE SHOULD MY APPLICATION GO?

- If you want an application, such as the search or language application, to be embedded on every site page, you should embed the application in the theme.
- In the case where you want to embed an application with a particular layout not to be displayed on every page, you can embed it in the layout template.
- This gives developers as much flexibility as they need when embedding applications.

Notes:

Chapter 5

Customizing the Front-end



TEMPLATES IN LIFERAY

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

USING TEMPLATES

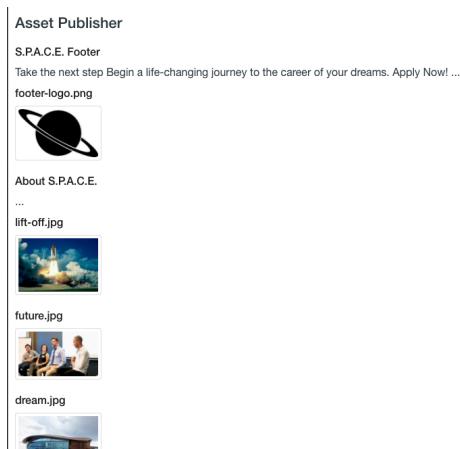
- ❖ As previously noted, you can provide styling for every aspect of the platform.
- ❖ Global Styling and page design can be modified using theme and layout templates.
- ❖ Liferay also provides *Application Display Templates (ADTs)* for styling different aspects of applications, which have integrated the ADT framework.

TEMPLATE EXAMPLES

- Templates in Liferay consist of *FreeMarker* templates you can use with different resources.
- There are a number of resources on the platform where you can use templates for styling.
 - **Assets:** *Web Content* and *Dynamic Data Lists* are special content types; they have their own templates.
 - **Applications:** A number of applications such as the *Asset Publisher*, or *Navigation*, can be styled through ADTs.
 - **Custom Applications:** Custom-developed applications can also have ADTs attached, giving you stylistic control.
 - **Notifications:** You can provide branding for *Workflow* email notifications.
- Additionally, you can use *Soy Templates* and *JSX* templates when developing applications.

ASSET PUBLISHER ADT EXAMPLES

- Let's take a look at some default Asset Publisher ADTs:



The screenshot shows the Asset Publisher ADT interface. At the top, there is a header with the title "Asset Publisher" and a sub-header "S.P.A.C.E. Footer". Below the header, there is a message: "Take the next step Begin a life-changing journey to the career of your dreams. Apply Now! ...". Underneath the message, there is a thumbnail for "footer-logo.png" which shows a black planet with a ring. Below the logo, there is a section titled "About S.P.A.C.E." followed by an ellipsis "...". There are two more thumbnails: "lift-off.jpg" showing a rocket launching and "future.jpg" showing three people sitting together. At the bottom, there is another thumbnail for "dream.jpg" showing a building.

FULL CONTENT VIEW ADT

- One of the options is a *Full Content* ADT that will display the full articles in list form.

The screenshot shows a Liferay Asset Publisher interface. At the top, there's a header with "Asset Publisher" and "S.P.A.C.E. Footer". Below this is a large image of a planet with a ring, followed by a text block: "Take the next step" and "Begin a life-changing journey to the career of your dreams." A blue "Apply Now!" button is at the bottom of the text block. Below the main content, there's a section titled "footer-logo.png" with a thumbnail image of the same planet logo. A "Info" button is visible, and below it, the file details: "footer-logo.png (Version 1.0)" and "Uploaded by Test Test, 10/28/16 5:53 PM". At the very bottom of the screenshot, the URL "WWW.LIFERAY.COM" and the Liferay logo are visible.

RICH SUMMARY ADT

- There is also a *Rich Summary* ADT that provides a *Read More* link with the ability to repost to Social Media.

The screenshot shows a Liferay Asset Publisher interface with a "Rich Summary" view. It displays several items, each with a thumbnail image and a "Read More" link. The items listed are:

- S.P.A.C.E. Footer: "Take the next step Begin a life-changing journey to the career of your dreams. Apply Now! ... Read More »"
- footer-logo.png: "Read More »"
- About S.P.A.C.E.: "... Read More »"
- lift-off.jpg: "Read More »"
- future.jpg: "Read More »"
- dream.jpg: "...

At the bottom of the screenshot, the URL "WWW.LIFERAY.COM" and the Liferay logo are visible.

USING TEMPLATES

- ❖ There are many other templates that can be used on the platform.
- ❖ Next, we'll take a look at using templates along with our theme to style our content.

Notes:



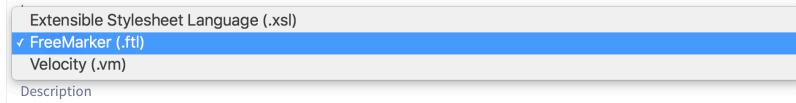
TEMPLATE LANGUAGE OPTIONS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

TEMPLATE LANGUAGE OPTIONS

- Templates can use the following templating languages:
 - **Velocity**: Velocity is a scripting language that lets you mix logic with HTML. Velocity is deprecated as of 7.0.
 - **Extensible Style Sheet Language**: Widely used for transforming XML into other formats.
 - **FreeMarker**: FreeMarker is a templating language that could be considered a successor to Velocity and is recommended.



FREEMARKER: THE WAY FORWARD

- FreeMarker templates in Liferay act as the intermediary between the back-end code in Java and the front-end.
- They enhance HTML by adding constructs such as variables, conditional statements, and loops.
- Once it's processed, the result is HTML, which is then styled by your CSS and displayed by the browser.
- FreeMarker provides a straightforward, clean, and simple method for incorporating dynamic content in a webpage.
- It permits the user to use a simple yet powerful template language to reference objects defined in the Java code.

FREEMARKER OPTIONS

- Here are some examples of what you can use in Freemarker:
 - **Variable:** accessed by its name \${var_name}
 - **Arithmetics:** +, -, /
 - **Logical Operators:** &&, ||, <=
 - **Sequence slice:** \${profile.assets[1..]}
 - **Include files:** [#include "header.html"]
 - **Comments:** [#- this is a comment -]
- You can also use if else statements or loops, using the directive symbol #.
- Macros can also be created and reused using @.
- Let's walk through some examples.

FREEMARKER EXAMPLES IN THE THEME

- Let's look at some examples from the theme we created.

- We defined a variable in the `init_custom.ftl` file.

```
<#assign show_header_search = getterUtil.getBoolean(themeDisplay.  
getThemeSetting("show-header-search")) />
```

- The variable can then be used in the following way:

```
 ${show_header_search}
```

- Our theme also has an if statement that includes a file if a theme setting is turned on.

```
<if has_navigation>  
<include "$full_templates_path}/navigation.ftl" />  
</if>
```

- We also have macros being used for things like the *menu*.

```
<@liferay.control_menu />
```

USING TAGLIBS WITH FREEMARKER

- Developers using FreeMarker have access to Liferay's taglibs in templates.
 - There is no need to instantiate taglibs as they're already available.
- Taglibs are accessed by indicating the TLD file name with underscores.
 - For example, `liferay-portlet-ext.tld` is specified as `@liferay_portlet_ext`.
- Note: The `utilLocator`, `objectUtil`, and `staticUtil` variables for FreeMarker and the `utilLocator` variable for Velocity are disabled by default.
 - These variables are vulnerable to remote code execution and privilege escalation and should be used with caution, if enabled.

FREEMARKER EXAMPLES IN THE CONTENT TEMPLATES

- ❖ We can see an example of the Carousel Content Template.
- ❖ This section includes the <#list> as well in order to iterate through content for display in the Carousel format.

```
<#list contents.getsiblings() as cur_contents>
    <div class="${(cur_contents?counter == 1)?then('active', '')} item">
        <#assign article = cur_contents.getData()?eval />

        <!-- Here is our taglib call -->
        <@liferay_ui["asset-display"]
            className=article.className
            classPK=getterUtil.getLong(article.classPK, 0)
            template="full_content"
        />
    </div>
</#list>
```

- ❖ The <@liferay_ui["asset-display"]> is a taglib that allows us to grab information and display content articles.

ACCESSING DATABASE INFORMATION

- ❖ Although Freemarker gives us a helpful way to interact with back-end code, it doesn't natively provide access to contextual information we may need (e.g., current user, page information).
- ❖ In some cases, the need may arise to create custom variables that can access the database and inject contextual information into our templates.
- ❖ We can use the *serviceLocator* property to access the database, but would have to leave it unrestricted, which can become a security risk.
- ❖ *Context Contributors* offer a middle ground where we can safely access contextual information in our template.

CONTEXT CONTRIBUTORS

- *Context Contributors* give users the ability to inject contextual variables and functionality into Freemarker templates.
- There are two types of *Context Contributors* in Liferay DXP:
 - TYPE_GLOBAL: Variables that are available globally
 - TYPE_THEME: Variables available only within themes
- For more information on creating and using *Context Contributors*, documentation is available at https://dev.liferay.com/develop/tutorials/-/knowledge_base/7-0/context-contributors

CHANGES IN FREEMARKER FOR DXP

- There are some changes to Freemarker Templates in DXP that are worth noting for those who created templates on previous version.
- One major change is that the theme variable is no longer injected into the FreeMarker context.
- For more information about why the theme variable was removed for Liferay DXP and suggestions for updating your code, go here:
https://dev.liferay.com/develop/reference/-/knowledge_base/7-0/breaking-changes#taglibs-are-no-longer-accessible-via-the-theme-variable-in-freemarker

DEFAULT EXAMPLE

- With FreeMarker, our pages can go from this:

+ Asset Publisher :

Your dreams cannot be stopped
Your dreams cannot be stopped This is not the work of any one man or even a group of men Buy why, some say, the moon? Why choose this as our goal?...

When I orbited the Earth in a spaceship
When I orbited the Earth in a spaceship I saw for the first time how beautiful our planet is Mankind, let us preserve and increase this beauty, and...

We want to explore
We want to explore We're curious people We want to explore. We're curious people. Look back over history, people have put their lives at stake...

We have an infinite amount to learn
We have an infinite amount to learn both from nature and from each other We have an infinite amount to learn both from nature and from each other ...

We choose to go to the moon
We choose to go to the moon We have an infinite amount to learn both from nature and from each other You know, being a test pilot isn't always...

We are all connected
We are all connected To each other, biologically. To the earth, chemically Never in all their history have men been able truly to conceive of the...

WWW.LIFERAY.COM



TEMPLATE EXAMPLE

- To something like this:

+ Asset Publisher :

Your dreams cannot be stopped
This is not the work of any one man or even a group of men
BY

When I orbited the Earth in a spaceship
I saw for the first time how beautiful our planet is
BY

We want to explore
We're curious people
BY

We have an infinite amount to learn
both from nature and from each other
BY

We choose to go to the moon
We have an infinite amount to learn both from nature and from each other
BY

We are all connected
To each other, biologically. To the earth, chemically
BY

WWW.LIFERAY.COM



Notes:



USING LEXICON CSS COMPONENTS IN TEMPLATES

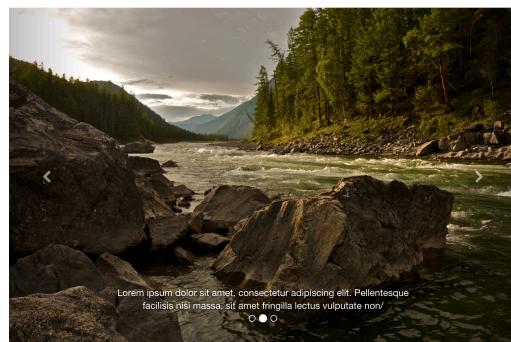
Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

LEXICON CSS COMPONENTS AND ELEMENTS

- ❖ As we discussed earlier, Lexicon CSS provides a large number of styles and components we can take advantage of in development.
- ❖ We've seen that we can use these styles and components in Themes.
- ❖ You can also use Lexicon CSS components in various Templates to include consistent and modern design for your applications and content.

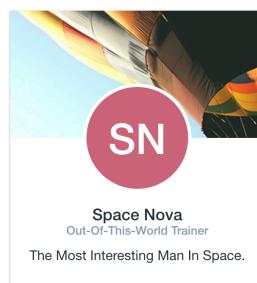


SIMPLIFYING TEMPLATE DEVELOPMENT

- ❖ Using the Lexicon CSS framework can simplify development across the board.
- ❖ In many cases, you can simply copy code snippets from the component you are interested in using, and it'll work in your Templates.
- ❖ Lexicon CSS components are also built with modularity in mind, further simplifying the development of consistent, modern content presentations.
- ❖ The list of Lexicon CSS components and elements can be found here:
<http://liferay.github.io/clay/>
- ❖ Let's take a look at some examples.

DISPLAYING USER INFORMATION WITH CARDS

- ❖ One example of a component we can use is a *Card* component.
- ❖ This component is versatile, as it can be combined with other components to provide the design developers need.
- ❖ All we need is a <div> element with the class `card`, and we can build from there.
- ❖ Let's take a look at the code behind this example.



CARD-CODED

- ❖ In our example, we're actually using a couple Lexicon CSS components and elements to provide information about our User.
- ❖ Inside the `card` component, we are using the `user-icon` component.
- ❖ We're also using the `crop-img` component to fit the image to the Card.

```
<div class="card" style="max-width: 300px;">
    <div class="crop-img crop-img-bottom crop-img-center" style="height: 150px
        
    </div>
    <div class="user-icon user-icon-danger user-icon-xxl" style="border: 4px
        solid #FFF; line-height:120px; margin: -64px auto 0; position: relative;">
        <span>SN</span>
    </div>
    <div class="card-block" style="text-align: center;">
        <h3 style="margin:0;">Space Nova</h3>
        <h5 class="text-default" style="margin-top:0;">
            Out-Of-This-World Trainer</h5>
        <p>The Most Interesting Man In Space.</p>
    </div>
</div>
```

BADGES, LABELS, AND STICKERS, OH MY

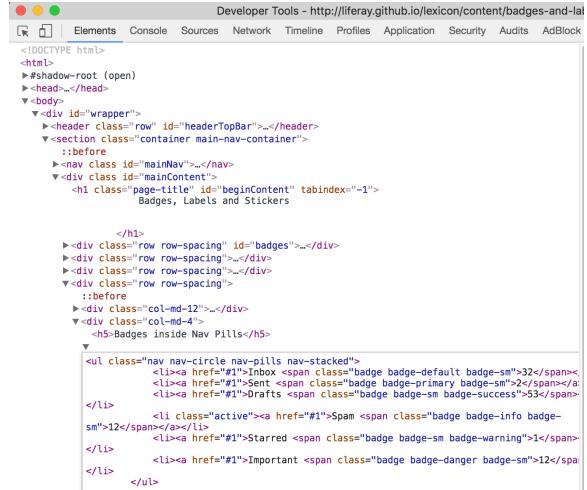
- ❖ *Badges, Labels, and Stickers* are other examples of smaller components.
- ❖ These components can be placed on larger components to maintain a consistent look-and-feel.
- ❖ For example, we could place a Sticker on an image using something like this:

```
<div class="aspect-ratio">
    
    <span class="sticker sticker-danger">PDF</span>
</div>
```



INSPECTING ELEMENTS

- ❖ One simple way to speed up the usage of these components is to take advantage of your browser's developer tools.
- ❖ In many cases, you can use *Inspect Element* to copy/paste the code into your Template and modify it as needed.



The screenshot shows the developer tools interface with the 'Elements' tab selected. The page title is 'Developer Tools - http://liferay.github.io/lexicon/content/badges-and-l...'. The code pane displays the HTML structure of a page, specifically focusing on a section titled 'Badges, Labels and Stickers'. The code includes various CSS classes like 'row', 'col-md-12', and 'col-md-4' used for layout, and badge-related classes such as 'badge badge-default', 'badge badge-primary', 'badge badge-success', 'badge badge-info', and 'badge badge-warning'.

```
<!DOCTYPE html>
<html>
  >#shadow-root (open)
    ><head>
      ><body>
        ><div id="wrapper">
          ><header class="row" id="headerTopBar"></header>
          ><section class="container main-nav-container">
            ><nav class="mainNav"></nav>
            ><div class="mainContent">
              ><h1 class="page-title" id="beginContent" tabindex="-1">
                Badges, Labels and Stickers
              </h1>
            ><div class="row row-spacing" id="badges"></div>
            ><div class="row row-spacing"></div>
            ><div class="row row-spacing"></div>
            ><div class="row row-spacing">
              ><div class="col-md-12"></div>
              ><div class="col-md-4">
                ><h5>Badges inside Nav Pills</h5>
                ><ul class="nav nav-circle nav-pills nav-stacked">
                  ><li><a href="#">Inbox <span class="badge badge-default badge-sm">32</span></a>
                  ><li><a href="#">Sent <span class="badge badge-primary badge-sm">2</span></a>
                  ><li><a href="#">Drafts <span class="badge badge-sm badge-success">53</span></a>
                  ></li>
                  ><li class="active"><a href="#">Spam <span class="badge badge-info badge-sm">12</span></a>
                  ><li><a href="#">Starred <span class="badge badge-sm badge-warning">1</span></a>
                  ></li>
                  ><li><a href="#">Important <span class="badge badge-danger badge-sm">12</span></a>
                  ></li>
                ></ul>
              </div>
            </div>
          </div>
        </body>
      </html>
```

THEMING COMPONENTS

- ❖ As we mentioned earlier, you can completely control the styling of each component.
- ❖ Each component can be customized to suit your needs with the *Lexicon CSS Customizer* or through the component CSS in the Theme.
- ❖ With this in mind, you can easily use the Lexicon CSS components and inherit the presentation from the theme itself.
- ❖ This means you can control styles and still maintain the simplicity and flexibility of Template development.

BEST PRACTICES FOR COMPONENTS AND ELEMENTS

- ❖ As you can see, there are many components and elements in Lexicon CSS.
- ❖ Some components overlap, some look the same at first glance, and several are combined to form a whole new component.
- ❖ It's always good to take the time to choose the right component for the job.
- ❖ As we look forward to Template development, keep these in mind.



Notes:

Chapter 6

Delivering Consistent Content Experiences



STYLING CONTENT WITH WEB CONTENT TEMPLATES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

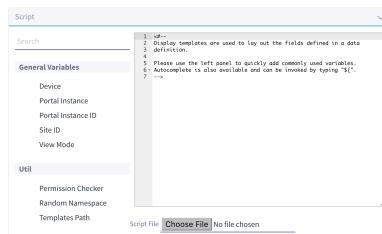
No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

STYLING CONTENT ARTICLES

- ❖ You can provide styles to *Web Content* using either the WYSIWYG editor directly, or by embedding the styles into a *Web Content Template (Template)*.
- ❖ The Alloy Editor can be used to add HTML/CSS/JavaScript changes into basic Web Content.
- ❖ We're going to focus on the more advanced Template option.
- ❖ A Template is responsible for displaying content from a *Web Content Structure (Structure)* and ultimately determines how the content is displayed.

ADDING TEMPLATES

- ❖ In *Site Administration*→*Content*→*Web Content*, you can add Structures and Templates.
- ❖ Templates are managed in the *Options* menu.
- ❖ You have two methods of adding templates:
 - ❖ Attaching an *ftl* file
 - ❖ Using the editor
- ❖ The editor gives you quick access to some default variables.



The screenshot shows the Liferay Script editor. The top navigation bar has tabs for 'Script', 'Search', and 'Script File'. Below the tabs is a sidebar with sections for 'General Variables' (Device, Portal Instance, Portal Instance ID, Site ID, View Mode) and 'Util' (Permission Checker, Random Namespace, Templates Path). The main area contains a code editor with the following content:

```
1<@-->
2<@-->
3<@-->
4<@-->
5<@-->
6<@-->
7<@-->
8<@-->
9<@-->
10<@-->
11<@-->
12<@-->
13<@-->
14<@-->
15<@-->
16<@-->
17<@-->
18<@-->
19<@-->
20<@-->
21<@-->
22<@-->
23<@-->
24<@-->
25<@-->
26<@-->
27<@-->
28<@-->
```

Below the code editor is a 'Choose File' button and a message 'No file chosen'.

WWW.LIFERAY.COM

LIFERAY.

DEVELOPING TEMPLATES

- ❖ Web Content is unique in that Structures can be provided by *Content Managers*.
- ❖ When developing Templates, you can simply reference the Structure fields and determine how to display them.
- ❖ All HTML tags, by default, will inherit the global styling from the Theme.
- ❖ You can use the `<style>` and `<script>` tags to provide custom CSS and JavaScript for the Template.



The screenshot shows the Liferay Script editor. The top navigation bar has tabs for 'Script', 'Search', and 'Script File'. Below the tabs is a sidebar with sections for 'General Variables' (Device, Portal Instance, Portal Instance ID, Site ID, View Mode) and 'Fields' (Title, Variable:title, Class:TemplateNode, Heading 1). The main area contains a code editor with the following content:

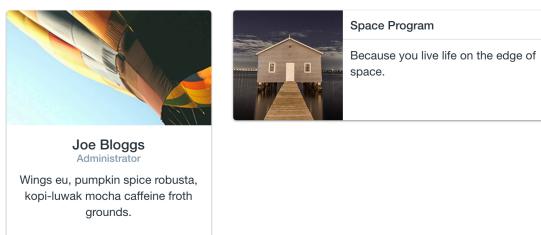
```
1<@-->
2<@-->
3<@-->
4<@-->
5<@-->
6<@-->
7<@-->
8<@-->
9<@-->
10<@-->
11<@-->
12<@-->
13<@-->
14<@-->
15<@-->
16<@-->
17<@-->
18<@-->
19<@-->
20<@-->
21<@-->
22<@-->
23<@-->
24<@-->
25<@-->
26<@-->
27<@-->
28<@-->
```

WWW.LIFERAY.COM

LIFERAY.

LEXICON CSS IN TEMPLATES

- ❖ As discussed previously, Lexicon CSS styles can easily be applied to both Themes and Templates.
- ❖ Developers can take advantage of Lexicon CSS components and elements to anything from *Cards* to *Modals*.
- ❖ If the Theme has customized any of the base Lexicon CSS styling, these changes will also apply to Templates.



GENERIC TEMPLATES

- ❖ You can also create what are called *Generic Templates*.
- ❖ *Generic Templates* are Web Content Templates that are not associated with a Structure.
- ❖ This allows you to create stand-alone Templates that include certain taglibs, and/or embedded applications, that can be applied to any content.
- ❖ Generic Templates can also be imported into other Templates using:
`<#include "${templatesPath}/TEMPLATE-ID" />`

EMBEDDING APPLICATIONS IN TEMPLATES

- Applications, custom or core, can be embedded into Templates.
- This means you can ensure application functionality is displayed with Web Content.
- For example, if we wanted to embed the *Currency Converter* application, we could do the following:

- In FreeMarker:

```
<@liferay_portlet_ext["runtime"] portletName="com_liferay_currency_
_converter_web_portlet_CurrencyConverterPortlet" />
```

- In Velocity:

```
$theme.runtime("com_liferay_currency_converter_
web_portlet_CurrencyConverterPortlet");
```

THE POWER OF TEMPLATES

- You can provide consistent look and feel on the platform with Web Content Templates.
- These Templates have access to many of the same tools used in front-end development such as:
 - Bootstrap
 - CSS
 - HTML
 - Lexicon CSS
 - Freemarker

Notes:



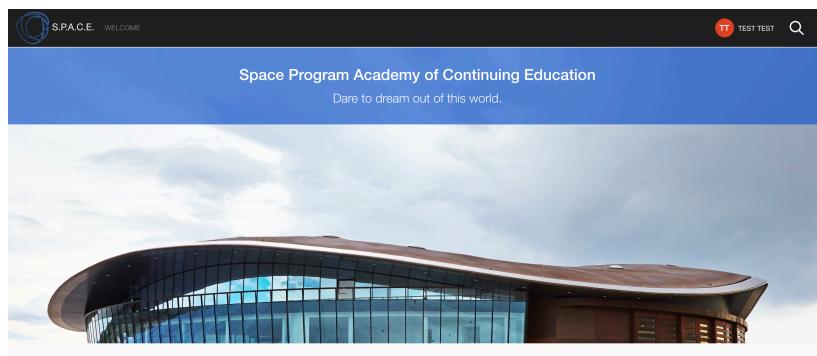
CREATING CONTENT TEMPLATES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

EXAMINING WEB CONTENT TEMPLATES

- ❖ Content creators are typically responsible for Structures, while front-end developers are responsible for Templates.
- ❖ Let's first examine the Template included in our Theme.
- ❖ Afterwards, we will create some new Templates together.



EXERCISE: CATCHING UP

- If you don't currently have the completed Theme installed:
 1. **Copy** the provided space-program-theme.war from the folder `exercises/front-end-developer-exercises/06-web-content-templates`.
 2. **Paste** the file in Liferay's deploy/ folder.
 3. **Open** the Menu.
 4. **Go to** Site Administration→Navigation.
 5. **Click** on the Options menu→Configure for Public Pages.
 6. **Click** Change Current Theme.
 7. **Click** on the Space Program theme to select it.
 8. **Click** Save.
 9. **Close** the dialog.

STYLING THE PYGON STRUCTURE

- Embedded in our theme is the *Polygon* Structure and Template.
- The Template is styling the following fields as defined by the Structure:
 - **Cover Image:** A *documents and media* field to select an image
 - **Title:** A *text* field for the image title
 - **Sub Title:** A *textarea* field adding subtitle information
 - **Content:** A *ddm-text-html*, otherwise known as the *Alloy Editor*, for providing additional content

Fields

Cover image

Title

Sub Title

Content

THE PYGON TEMPLATE

- ❖ Let's look at a couple key points in the Template.
- ❖ The variables are assigned at the top of the file:

```
<#assign
    author = request.attributes.author!=""
    displayMode = getterUtil.getInteger(request.attributes.displayMode!0,0)
    viewURL = request.attributes.viewURL!=""
/>
```

- ❖ We also see the Template controlling different display modes:

```
<if displayMode == 1 >
...
<elseif displayMode == 2 >
...
<elseif displayMode == 3 >
...
<else>
...
</if>
```

STYLING STRUCTURE FIELDS

- ❖ The Structure fields can be added to the template with the following syntax:
 `${[field-name].getData()}`
- ❖ The Structure fields being styled in the *Polygon Template* are as follows:
 - ❖ coverImage.getData()
 - ❖ title.getData()
 - ❖ subTitle.getData()
 - ❖ content.getData()

PORYGON TEMPLATED CONTENT ON DISPLAY

- Once everything is saved and content added, we see something like this.



When I orbited the Earth in a spaceship

I saw for the first time how beautiful our planet is

Mankind, let us preserve and increase this beauty, and not destroy it!

As I stand out here in the wonders of the unknown at Hadley, I sort of realize there's a fundamental truth to our nature, Man must explore . . . and this is exploration at its greatest.

I believe every human has a finite number of heartbeats. I don't intend to waste any of mine

STYLING FOR THE LANDING PAGE

- The S.P.A.C.E. content team has created some Structures and content for the website landing page.
- We are responsible for providing *Web Content Templates* that meet the landing page business requirements.
- Design requirements are as follows:
 - Banner:** The banner will include an image with text overlay.
 - News Feed:** The news feed will contain a self-updating feed of news for S.P.A.C.E.
 - Mid-Page Content:** This will be a 3-column content article with images and text overlay.
 - Events Feed:** The news feed will contain a self-updating feed of upcoming events for S.P.A.C.E.
 - Footer:** The footer content will contain an image, text overlay, a logo, and action button to apply.
- First, let's walk through creating a banner template for the S.P.A.C.E. Site.

EXERCISE: PREPARING FOR TEMPLATE CREATION

- First, let's import the content from the content team.
- 1. **Open** the *Menu*.
- 2. **Go to** *Content→Web Content* in the Site Administration panel.
 - Make sure you're logged in and the S.P.A.C.E. Site is selected.
- 3. **Open** the *Options* menu.
- 4. **Choose** *Export/Import*.

EXERCISE: IMPORTING CONTENT

1. **Click** on the *Import* tab.
 2. **Choose** the space-content-7.0.1ar from
exercises/front-end-developer-exercises/ 06-web-content-templates.
 3. **Click** *Continue→Import*.
 4. **Close** the pop-up.
- ✓ Now we are ready to create our templates!

EXERCISE: VIEWING THE DEFAULT

1. **Click** Go to Site in the Site Administration panel in the *Menu*.
2. **Open** the Options menu for the *Hello World* application.
3. **Click** Remove.
4. **Click** OK on the browser pop-up.
5. **Open** the Add menu at the top right.
6. **Open** the Content section.
7. **Drop** the *S.P.A.C.E. Banner*, *About S.P.A.C.E.*, and *S.P.A.C.E. Footer* articles on the page.



WWW.LIFERAY.COM

 LIFERAY.

DEFAULT STRUCTURE AND TEMPLATE

- ❖ Each article is displaying the default template that simply includes the structured content.
- ❖ To accomplish the business requirements, we'll need more control over how the Structure is presented.
- ❖ Let's start with modifying the banner Template.

WWW.LIFERAY.COM

EXERCISE: MODIFYING THE BANNER TEMPLATE

- Let's provide a template that takes our text and places it over the image.
 1. **Open** the *Menu*.
 2. **Go to** *Content→Web Content* in the Site Administration Panel.
 3. **Open** the *Options* menu.
 4. **Choose** *Templates*.
 5. **Click** on the *1 Column Header Template* and scroll down to the editor.
 6. **Click** *Choose File* under the editor.
 7. **Choose** *1-column-banner.ftl* found in
exercises/front-end-developer-exercises/
06-web-content-templates/template-src.
 8. **Click** *Save*.

✓ Now we have our responsive banner!

THE BANNER FTL IMAGE

- Our banner is a simple FreeMarker template.
- We are simply providing a `<div>` with the *figure* class that includes our image and text overlay.
- Our image is using the default editor option for an image field, but we have added `crop-img-top` class with a specified height.
- By default, the editor adds the `img-responsive` class to our images to ensure responsive design.
- For more on Lexicon CSS figure design, you can go here:
<https://liferay.github.io/clay/content/figures/>

```
<div class="crop-img-top" style="height: 500px">
  
</div>
```

THE BANNER FTL TEXT

- ❖ Our text has been centered at the top of the image using the `figcaption` and `center` classes.
- ❖ By including the `flex` classes, we've also ensured our text is responsive.
- ❖ Inside the `<div>` structure, we have our title and heading using our `getData()` methods.

```
<div class="figcaption-top figcaption-info">
    <div class="flex-container" style="height: 100%">
        <div class="flex-item-center text-center" style="width: 100%">
            <h1>${title.getData()}</h1>
            <p class="lead">${heading1.getData()}</p>
        </div>
    </div>
</div>
```

S.P.A.C.E. BANNER COMPLETE



EXERCISE: MODIFYING THE ABOUT S.P.A.C.E. TEMPLATE

- ❖ Next, let's modify our *3 Columns with Images* template to make sure the *About S.P.A.C.E.* article displays as expected.
1. **Click** on the *3 Columns with Images Template* and scroll down to the editor.
 2. **Click** *Choose File* under the editor.
 3. **Choose** the *3-column-with-images.ftl* found in *exercises/front-end-developer-exercises/06-web-content-templates/template-src*.
 4. **Click** *Save*.
- ✓ Now our mid-page content will display in 3 columns with text overlaying images.

THE 3 COLUMN WITH IMAGES FTL STRUCTURE

- ❖ The *3 Column with Images* Template is relatively similar to the Banner Template.
- ❖ We include the class `row` so we can take advantage of Bootstrap columns for responsive design.
- ❖ Each item in the row contains the following classes for mobile, tablet, and desktop breakpoints:

```
<li class="col-lg-4 col-md-4 col-xs-6">
```

THE 3 COLUMN WITH IMAGES COLUMNS

- ❖ Each column in this Template is the same, referencing different fields from the Web Content Structure.
- ❖ We're using the **figure** classes for our rounded styling.
- ❖ The text overlay is being accomplished using the **figcaption** class.

```
<div class="figure figure-rounded aspect-ratio aspect-ratio-3-to-2">
<if image1.getData()?? && image1.getData() != "">
    
    <div class="figcaption-full text-center">
        <div class="flex-container" style="height: 100%">
            <div class="flex-item-center">
                <h2>${heading1.getData()}</h2>
                <p>${content1.getData()}</p>
            </div>
        </div>
    </if>
</div>
```

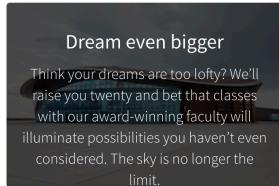
WWW.LIFERAY.COM



ABOUT S.P.A.C.E. COMPLETE

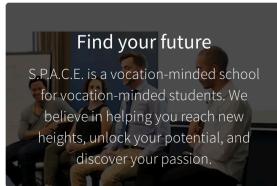
Dream even bigger

Think your dreams are too lofty? We'll raise you twenty and bet that classes with our award-winning faculty will illuminate possibilities you haven't even considered. The sky is no longer the limit.



Find your future

S.P.A.C.E. is a vocation-minded school for vocation-minded students. We believe in helping you reach new heights, unlock your potential, and discover your passion.



Prepare for lift-off

We don't just teach you—we equip you to take on the real-life challenges of being an... Astronaut? Aeronautics engineer? AI expert? You get to fill in the blank.



WWW.LIFERAY.COM



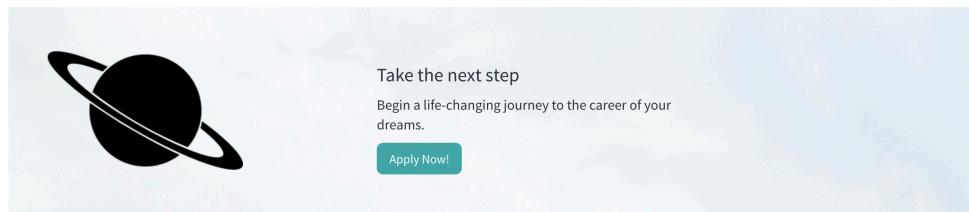
EXERCISE: MODIFYING THE FOOTER TEMPLATE

- Finally, let's modify the *Footer template* to display our text, image, and an action button.
1. **Click** on the *Footer Template* and scroll down to the editor.
 2. **Click** *Choose File* under the editor.
 3. **Choose** the *footer-content.ftl* found in *exercises/front-end-developer-exercises/06-web-content-templates/template-src*.
 4. **Click** *Save*.
- ✓ Now our footer is ready to go!

THE FOOTER FTL

- In the course of development, you'll likely come across the need to work with legacy code.
- In our footer example, we've worked with legacy HTML code to create the template.
- We're using a *style* tag to place the *bgimage* field from our Structure as the background.
 - This example actually uses the same image as the banner, but only uses the image clouds as the background padding class.
- Then, using a *<table>*, we can provide our columns to reference our logo, text, and action button.
- With that, we have accomplished the business requirements for web content on the front page of S.P.A.C.E.

S.P.A.C.E. FOOTER COMPLETE



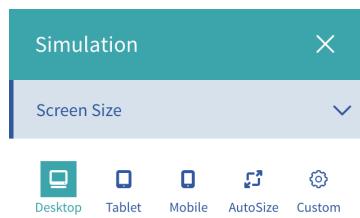
WWW.LIFERAY.COM

LIFERAY.

EXERCISE: TEMPLATE PREVIEW

- ❖ We can test our templates' responsive design by using the Simulator panel on the right-hand side of the page.
- ❖ Let's take a look at our template on different breakpoints.

1. **Open** the *Menu*.
2. **Click** *Go to Site* in Site Administration.
3. **Click** on the Simulation Menu at the top right of the Control Menu.



WWW.LIFERAY.COM

LIFERAY.

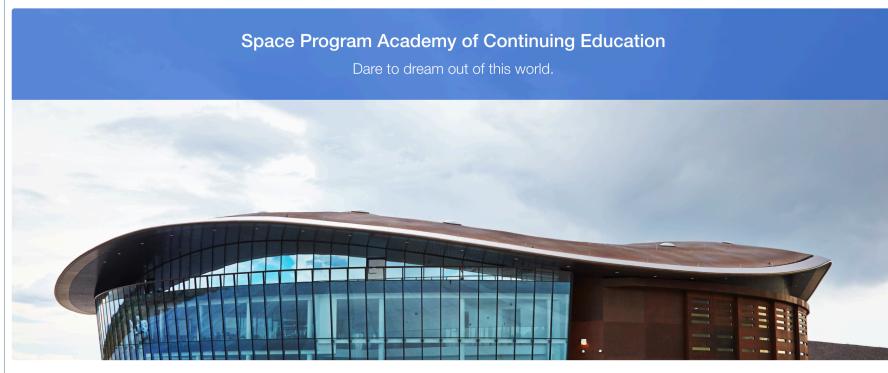
SIMULATION OPTIONS

- ❖ The Simulation Menu has options for following:
 - **Desktop:** This allows you to see content from the Desktop view
 - **Tablet:** This allows you to see the Vertical and Horizontal preview of a Tablet.
 - **Mobile:** This allows you to see the Vertical and Horizontal preview of a Mobile device.
 - **Auto-size:** This allows you to resize as needed.
 - **Custom:** This allows you to set custom breakpoints.
- ❖ Let's look at our Templates in the three main breakpoints.

PREVIEWING OUR TEMPLATES FOR DESKTOP

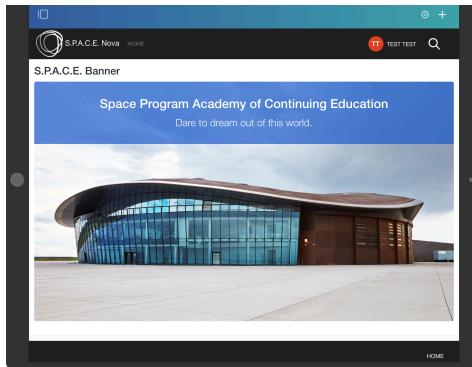
- ❖ We can view our Templates from a Desktop perspective with the default Desktop option.

S.P.A.C.E. Banner



EXERCISE: PREVIEWING OUR TEMPLATES FOR TABLET

1. Click on the *Tablet* option to view our templates as they'll display on a tablet.
 - Clicking on the Tablet option twice will toggle the vertical and horizontal view.



WWW.LIFERAY.COM

 LIFERAY.

EXERCISE: PREVIEWING OUR TEMPLATES FOR MOBILE

1. Click on the *Mobile* option to view our templates as they'll display on a mobile device.
 - Clicking on the Mobile option twice will also toggle the vertical and horizontal view.



WWW.LIFERAY.COM

 LIFERAY.

Notes:



USING TAGLIBS AND PREFERENCES IN TEMPLATES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

STYLE ALL THE CONTENT

- Content Templates can be extremely powerful.
- Our current templates take content from a Structure and style it in a consistent way.
- We've used Lexicon CSS to create beautiful content displays like a banner and thumbnails.
- As we style more content throughout the site, we may need more advanced tools:
 - Taglibs
 - Repeatable fields from Structures
 - Passing settings between templates

S.P.A.C.E. NEEDS A PLACE

- The Content Team in S.P.A.C.E. needs a dedicated section to show upcoming events.
- Between incoming student orientations, tours, summer sessions, field trips, and career fairs, anyone could get lost in all that information.
- They'd like to organize the events so that:
 - It's easy to see the most recent events.
 - Important events are featured and stand out.
 - Events are created once, but restyled in different places.

CREATING AN EVENTS PAGE

- They've decided to create a new *Events* page on the site.
- Events will be created through Web Content and displayed in a number of different styles.
- Our Content Team has already created a Structure for events, but needs to style it.
- Some events will be featured, or highlighted, on the *Events* page.
- Instead of creating multiple Structures for events (whether they're featured or not), the Content Team has come up with something a little different.

THE COMPOSITE SOLUTION

- The Content Team has created an additional Structure, *Content List*.
- The *Content List* Structure contains:
 - A *ddm-journal-article* field called **contents**
- This field is different from what we saw before because it:
 - Allows the writer to choose a Web Content Article for the field
 - Can be repeated many times

WWW.LIFERAY.COM

LIFERAY.

DEVELOPING A NEW TYPE OF TEMPLATE

- Using the structured content from the Content Team, we'll need to:
 - Develop a Template for the *Event* Structure
 - Develop a Template for *Content List* that embeds other Web Content Articles
 - Develop Templates to display the *Content List* in different ways

EXERCISE: IMPORTING EVENTS

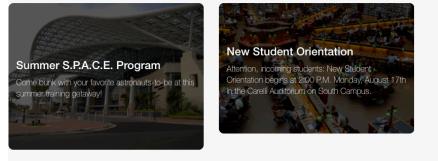
- The S.P.A.C.E. Content Team has already provided the Structures for us to build from.
1. **Open** the Menu.
 2. **Go to** *Content→Web Content* in the Site Administration panel.
 3. **Open** the *Options* menu.
 4. **Choose** *Export/Import*.
 5. **Click** on the *Import* tab.
 6. **Choose** the 02-space-events.lar from *exercises/front-end-developer-exercises/ 06-web-content-templates*.
 7. **Click** *Continue→Import*.
 8. **Close** the pop-up once it's successful.

STYLING EVENTS

- Our first task is to provide basic styling for the *Events*.
- We'll build a nice display with Lexicon CSS components, like *Card*, with the Structure fields.
- *Events* contain:
 - **Cover Image:** A beautiful image to display when listing or displaying the event
 - **Headline:** The name of the event
 - **Date:** When the event is happening
 - **Lead Text:** Brief information to display about the event
- We can style these into a nice card, displaying all the event details.

EXERCISE: BASIC EVENT STYLING

1. **Open** the *Options* menu in *Content→Web Content*.
 2. **Choose** *Templates*.
 3. **Click** on the *Event Display* template and scroll down.
 4. **Click** *Choose File* under the editor.
 5. **Choose** the *event-basic-display.ftl* found in
exercises/front-end-developer-exercises/
06-web-content-templates/template-src.
 6. **Click** *Save*.
- ✓ Now we have our styled events!

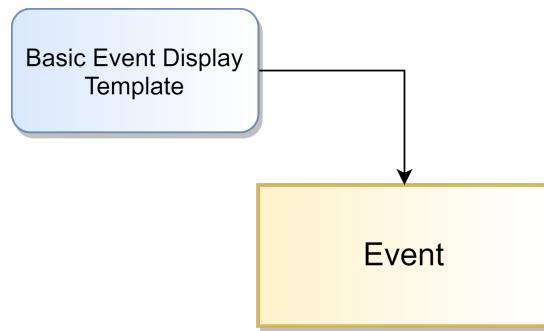


WWW.LIFERAY.COM

 LIFERAY.

THE EVENT DISPLAY TEMPLATE

- ❖ Events will now be styled using the basic styling outlined in the *Event Display* template.



WWW.LIFERAY.COM

 LIFERAY.

FEATURING EVENTS

- Now that we have our basic events styled, any time we want to show an event, it will look clean and consistent.
- The Content Team would also like to feature some events.
- A *Content List Structure* allows for selecting Web Content Articles to feature.
- We'll need to write a Template that can:
 - Embed content from another Web Content Article
 - Style multiple fields from a list
- To embed Web Content Articles, we'll need to use some other Liferay features.

EMBEDDING WEB CONTENT

- If you tried to use `getData()` on each Web Content Article, you'd only get:
 - The *Java class* that article uses
 - The *primary key* of the article
- If you're a Java developer, or have access to the `serviceLocator`, this might be useful.
- You could call the service for the Web Content Article (`JournalArticleService`) and ask for its content.
- But what if we can display articles *without* needing to call more back-end services?

USING TAGLIBS IN TEMPLATES

- ❖ Taglibs provide reusable components that simplify complex displays in Liferay.
- ❖ Normally, Java developers can use these in JSPs to make building application views easy.
- ❖ Templates have access to Taglibs using a simple syntax in FreeMarker:
`<@taglib_name["tag-name"] attribute=value />`
- ❖ There are a lot of taglibs available for displaying data and building complex UIs:
 - liferay_ui: some general components for displaying data
 - liferay_frontend: contains some Lexicon CSS components
 - aui: components for AlloyUI
- ❖ More information can be found in the appendix.

DISPLAYING CONTENT WITH A TAG

- ❖ Taglibs are so useful that applications like the *Asset Publisher* use them for displaying Web Content and other assets.
- ❖ Can we use the same tags as the *Asset Publisher* for displaying Web Content?
- ❖ The taglib `liferay_ui` contains the tag `asset-display` that takes three attributes:
 - `className`: the *Java class* of the asset
 - `classPK`: the *primary key* of the asset
 - `template`: how to display the asset
- ❖ We already have most of this information from our Structure field, contents.

DISPLAYING OUR ASSET LIST

- Using this tag, we can easily display the Web Content Article in FreeMarker:

```
<@liferay_ui["asset-display"]
    className=article.className
    classPK=getterUtil.getLong(article.classPK, 0)
    template="full_content"
/>
```
- The template called `full_content` just needs to render our Web Content Article as if it were being displayed on its own.
 - This way, the Content Template for the article gets used.
- We can use the tag `<@liferay_ui["asset-display"] \>` to display each article in our list.

USING REPEATABLE FIELDS

- Our list of Web Content Articles is implemented as a *repeatable field* in the Structure.
- When content creators are building an article, they can click a button to add as many copies of the field as they want.
- We could have 1, 2, or 10 articles, depending on how many fields there are.
- To use a list of fields, we need to be able to:
 - Get the list
 - Iterate over the list
- This is actually pretty easy to do.

GETTING THE LIST OF FIELDS

- ❖ When we want data from a field, we usually call its `getData()` method.
- ❖ Repeatable fields have *siblings*.
- ❖ Each *sibling* is a copy of the field.
- ❖ Each field would have a `getData()` method.
- ❖ In our case, each *sibling* would have the `className` and `classPK` for the article.
- ❖ Getting our list of content is pretty easy:
`$contents.getsiblings()`
- ❖ So how do we iterate over the list?

LIST BY LIST

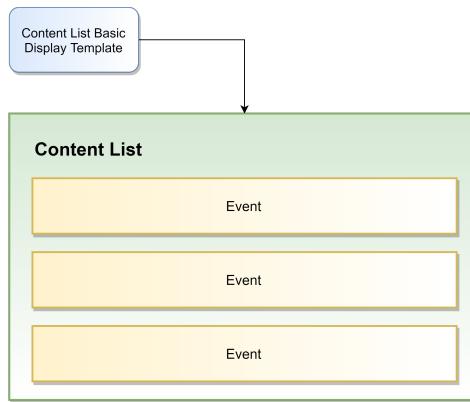
- ❖ We see list iteration in a number of our templates, using the directive `<#list>`.
- ❖ Listing items means we need to know the name of the *list* and each individual *item*:
`<#list items as item>
$item.getData()
</#list>`
- ❖ *items* is the list of variables, and *item* represents each item in the list.
- ❖ We can apply this to our articles:
`<#list contents.getsiblings() as cur_contents>
$cur_contents.getData()
</#list>`
- ❖ We now have enough information to create a template that can *iterate over a list* of Web Content Articles and *display each article*.

EXERCISE: DISPLAY A CONTENT LIST

1. **Click** on the *Content List Display* template and scroll down to the editor.
 2. **Click** *Choose File* under the editor.
 3. **Choose** the *content-list-basic-display.ftl* found in *exercises/front-end-developer-exercises/06-web-content-templates/template-src*.
 4. **Click** *Save*.
- ✓ Now we can show repeatable content.

CONTENT LIST DISPLAY TEMPLATE

- ❖ With the *Content List* structure, we can feature multiple Web Content Articles.
- ❖ We've set the basic styling in the *Content List Display* template.

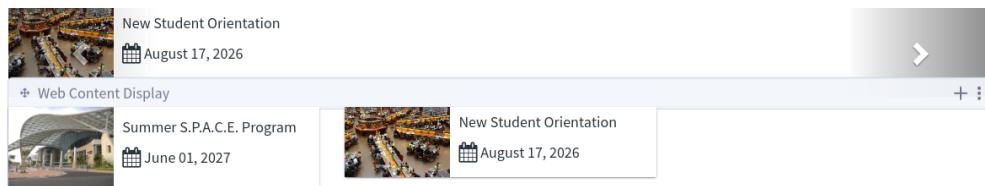


BUILDING DIFFERENT DISPLAYS

- It's really useful to be able to reuse the same Web Content Articles and just display them differently.
- We'd like to be able to provide multiple displays for featured events:
 - Basic display
 - Thumbnail display
 - Carousel display
- We've just completed the basic display, and could create new templates to build out a thumbnail and carousel display.

A CAROUSEL AND A THUMBNAIL

- What would it look like if we built carousel and thumbnail templates and displayed the Web Content Articles inside them?
- Not pretty.
- We need to change how the article displays in each component.



SETTING VARIABLES ACROSS TEMPLATES

- The *Events Display* template is responsible for how the article is displayed.
- But it doesn't know anything about how the carousel or thumbnail display or what changes they need.
- We need a way to *tell* the events template *how* to change its display for each template.

USING PREFERENCES TO SHARE SETTINGS

- All of our Web Content Articles are being displayed in the *Web Content Display* application.
- Each application has a way to store *preferences*, or settings, as variables.
- Setting *preferences* allows those settings to be used by any other template in the application.
- These preferences are available through an object in FreeMarker called \$freeMarkerPortletPreferences.
- We can set and get different values using this object:

```
<#assign VOID = freeMarkerPortletPreferences.setValue("mySetting", "stuff") />
...
<#assign mySetting = freeMarkerPortletPreferences.getValue("mySetting") />
```
- **Note:** we use an empty variable called \$VOID to run the method silently. We wouldn't want return values displayed on the page.

A CAROUSEL, A LIST, AND THUMBNAILS...

- ❖ We can now design each of the *Web Content Templates* to set a setting we'll call `view`:
 - `thumbnailListView`: We'll set `view` to this when we're displaying thumbnails.
 - `carouselListView`: We'll set `view` to this when we're displaying a carousel.
- ❖ When we're displaying the content in full, we won't set the `view` setting.
- ❖ Now, let's build each template to tell the *Event Display* template which style we need.

EXERCISE: BUILDING THUMBNAILS

1. **Click** the *Content List Thumbnail Display* template and scroll down to the editor.
 2. **Click** *Choose File* under the editor.
 3. **Choose** the `content-list-thumbnail-display.ftl` found in `exercises/front-end-developer-exercises/06-web-content-templates/template-src`.
 4. **Click** *Save*.
- ✓ Now we have a new display showing thumbnails of featured events!

THUMBNAIL DISPLAY TEMPLATE

- We've set the `thumbnailListView` setting in the *Thumbnail Display* template.
- This setting will be referenced in logic we'll add to the *Event Display* template.

Content List Thumbnail
Display Template

```
<#assign VOID =  
freeMarkerPortletPreferences.setValue("view",  
"thumbnailListView") />
```

EXERCISE: BUILDING A CAROUSEL

1. **Click** on the *Content List Carousel Display* template and scroll down to the editor.
 2. **Click** *Choose File* under the editor.
 3. **Choose** the `content-list-carousel-display.ftl` found in `exercises/front-end-developer-exercises/06-web-content-templates/template-src`.
 4. **Click** *Save*.
- ✓ Now we have a new display showing a carousel of featured events!

CAROUSEL DISPLAY TEMPLATE

- We've set the `carouselListView` setting in the *Carousel Display* template.
- This will be referenced in logic we'll add to the *Event Display* template.

Content List Carousel
Display Template

```
<#assign VOID =  
freeMarkerPortletPreferences.setValue("view",  
"carouselListView") />
```

MAKING OUR EVENT DISPLAY SMART

- We're going to smarten up our *Event Display* template so that it responds to the value of the `view` setting.
- Using `<if>` directives, we can build our event display using the right Lexicon CSS components for the containing template.
- If there is no `view` setting, then we'll just display the event the way we originally did.

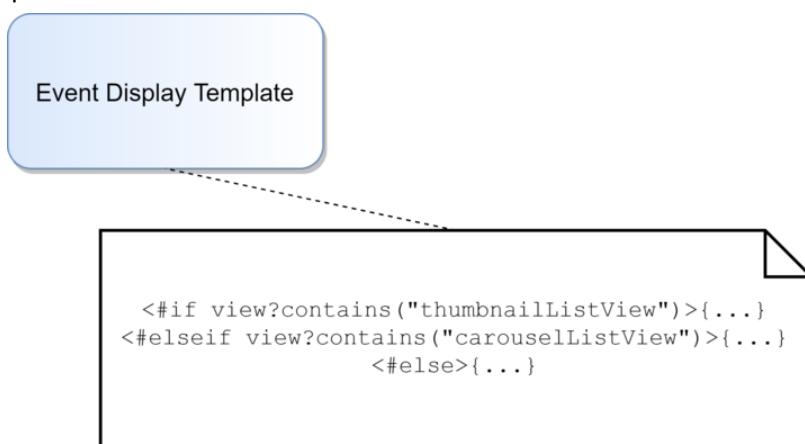
EXERCISE: SHARING IS CARING

1. **Click** on the *Event Display* template and scroll down to the editor.
2. **Click** *Choose File* under the editor.
3. **Choose** the *event-advanced-display.ftl* found in *exercises/front-end-developer-exercises/06-web-content-templates/template-src*.
4. **Click** *Save*.

✓ Now the events will display differently in each list type!

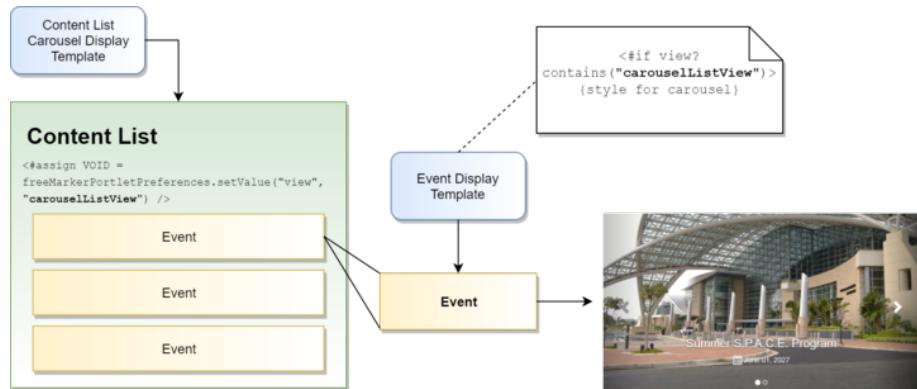
ADVANCED EVENT DISPLAY TEMPLATE

- ❖ In our updated *Event Display* template, we've added logic to allow the template to respond to the view preference set in the Content List templates.



TEMPLATES IN ACTION

- With the logic we've added, the *Event Display* template will respond to whichever template the Content List structure is using.



DISPLAYING LOTS OF CONTENT

- Now that our displays are smart, respond to different environments, and use the same type of Web Content (*Event*), let's import some events.
- Our Content Team has provided us with events that are featured on the new *Events* page.
- Let's update our site to display these events using our new templates.

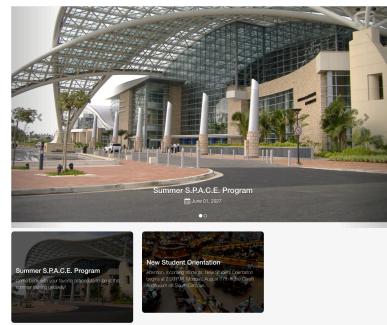
EXERCISE: IMPORTING EVENTS

1. **Go to** *Menu→Site Administration→Publishing.*
 2. **Click** on *Import*.
 3. **Click** *Add* at the bottom right.
 4. **Choose** the *SPACE_Events_Page_Complete.lar* from *exercises/front-end-developer-exercises/ 06-web-content-templates*.
 5. **Click** *Continue→Import*.
- ✓ Now we have content on display with our templates!

EXERCISE: BRAVE NEW TEMPLATES

1. **Open** the *Menu*.
 2. **Click** *Go to Site* in Site Administration.
 3. **Click** *Events* in the navigation.
- ✓ Now we can see our new events displays!

➤ **Note:** Templates may look different depending on the browser.



EXERCISE: APPLYING THE THEME

- ❖ When we imported our events, it reverted the theme selection.
 - ❖ Let's go ahead and re-apply the theme.
1. **Go to** *Menu→Site Administration→Navigation*.
 2. **Open** the *Options* menu next to *Public Pages*.
 3. **Click** *Configure*.
 4. **Click** *Change Current Theme*.
 5. **Select** the *Space Program Theme*
 6. **Click** *Save*.
- ✓ Now our theme is applied!

Notes:

Chapter 7

Customizing Workflow Email Notifications



CUSTOMIZING NOTIFICATION TEMPLATES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

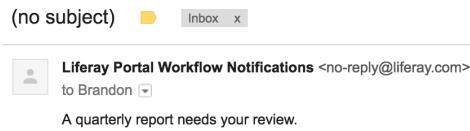
No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WORKFLOW OVERVIEW

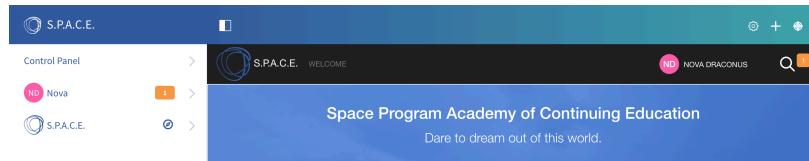
- ❖ Liferay's Kaleo Workflow engine makes it easy to build and use review processes with notifications to different parties.
- ❖ While front-end developers aren't responsible for setting up workflow, they can provide FreeMarker notification templates.
- ❖ These templates are useful for both branding and functionality.

WORKFLOW NOTIFICATIONS

- ▶ Notifications can be sent to different parties on the following channels:
 - ▶ **Email:** Email notifications sent out to those assigned to a workflow task.



- ▶ **User Notification:** Notifications sent to users logged into the platform



- ▶ **Instant Messenger & Private Message:** Placeholders for Social Office

NOTIFICATIONS IN A WORKFLOW DEFINITION

- ▶ Workflow Definitions are written in XML and include the notifications within.
- ▶ Here is an example of a review notification:

```
<notification>
    <name>Review Notification</name>
    <template>
        ${userName} sent you a ${entryType} for review in the workflow.
    </template>
    <template-language>freemarker</template-language>
    <notification-type>email</notification-type>
    <notification-type>user-notification</notification-type>
    <execution-type>onAssignment</execution-type>
</notification>
```

- ▶ In this simple example, we are using the \${userName} and \${entryType} variables to pull in information for the name of the submitter and the type of asset.

CREATING ADVANCED NOTIFICATION TEMPLATES

- ❖ Because Workflow notifications are just FreeMarker templates, front-end developers can provide the same kind of customization here that they would elsewhere.
- ❖ There are a number of context variables available that allow developers to add functionality.
- ❖ Lexicon CSS styling and JavaScript can also be added into the templates using the `<style>` and `<script>` tags.
- ❖ With these features at the ready, notifications can be as simple or advanced as the business requirement needs.

Notes:



GETTING INFORMATION FROM WORKFLOW

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

THE WORLD IS OUR LIBRARY

- ❖ Workflow definitions allow us to use FreeMarker natively in the workflow definition xml file.
- ❖ This means we can pull information from the platform.
- ❖ Using FreeMarker, let's see how to pull information from Liferay by modifying an existing workflow.

GETTING INFORMATION INTO THE WORKFLOW TEMPLATE

- ❖ Pulling information into our workflow definition is not much different from the same process for an Application Display Template.
- ❖ In this example, we'll start off simple and pull information from the workflow.
- ❖ Recall that as something goes through the workflow, workflow approvers are able to leave comments along the way.
- ❖ We'll pull information from the comments that are left from the workflow approvers.

WORKFLOWS IN S.P.A.C.E.

- ❖ S.P.A.C.E has been using the workflow process to enable instructors to grade assignments and leave feedback.
- ❖ Students aren't always going to be signing in and out of the S.P.A.C.E. platform.
- ❖ We want a way to ensure that students get feedback from their instructors via email.
- ❖ This exercise will take the comments left by the instructor and send them to the student through an email notification that we configure in the workflow definition.

EXERCISE: INCLUDING SNIPPETS

- Before we start modifying our workflow definition, we have some snippets we can use with *Brackets*.
 1. **Open** Brackets.
 2. **Click** on the dropdown in the left-hand side bar.
 3. **Click** *Open Folder...*
 4. **Go to** *exercises/front-end-developer-exercises/07-workflow-templates*.
 5. **Choose** the snippets folder.
- ✓ Now that we have our snippets, let's start modifying our workflow definition.

EXERCISE: USING FREEMARKER NOTIFICATIONS

- The workflow definition xml file has already been created for us.
- We need to enable our workflow definition to use FreeMarker and then add our notification template.
 1. **Go to** *exercises/front-end-developer-exercises/07-workflow-templates/* in your file manager.
 2. **Open** the *assignment-grading.xml* file in *Brackets*.
 3. **Find** `<name>Graded Notification</name>` on line 59.
 - This is where we will add our FreeMarker notification.
 4. **Go to** `<template-language>text</template-language>` on line 65.
 5. **Replace** *text* with *freemarker*.

EXERCISE: SENDING OUT EMAIL NOTIFICATIONS

- ❖ As previously discussed, Workflow notifications can also be configured to receive emails.
 - ❖ Let's go ahead and enable email notifications.
1. **Go to** `<notification-type>user-notification</notification-type>` on line 66.
 2. **Replace** `user-notification` with `email`.
 3. **Save** the file.
- ❖ Now that we have our workflow definition all set up, let's add our notification template to reference information from the workflow process.

EXERCISE: RETRIEVING THE COMMENTS FROM OUR WORKFLOW

- ❖ Notification templates can be added inside a `<template>` tag in the workflow definition.
 - ❖ Developers simply need to find the `template` section in the workflow definition and place the notification template in the `<![CDATA[]]>` section.
1. **Open** the `o1-workflow-comment-retrieve` snippet.
 2. **Copy** the contents of the snippet.
 3. **Go to** the `<template>` tag on line 62 of the `assignment-grading.xml` file.
 4. **Paste** the contents of the snippet within the `<![CDATA[]]>` tag on line 63.
 5. **Save** the file.

WHAT'S GOING ON HERE?

- ❖ Our email notification template is simple.
- ❖ We first assign `taskComments` to the `comments` variable, allowing us to display comments from the workflow process.
- ❖ Then we create a simple HTML structure that shows the graded assignment, using the `entryType` variable, and the included comments if they exist.

```
<![CDATA[  
  <#assign comments = taskComments!"">  
  <!-- email body -->  
  <p>  
    Your assignment, ${entryType}, has been graded by the instructor.  
  <if comments != "">  
    <br />Here are the comments included: <strong>$comments</strong>  
  </if>  
  </p>  
  <!-- signature -->  
  <p>Sincerely,<br /><strong>S.P.A.C.E. Instructor</strong></p>  
>]]>
```

STYLING AND BRANDING OUR EMAIL NOTIFICATIONS

- ❖ We can provide additional company branding or styling to our notification templates.
- ❖ Adding styling to the template is simple, but more complex notifications could potentially be a bit unwieldy.
- ❖ Instead of adding all the styling in the template, we can do things like reference web content in our notifications.
- ❖ Injecting web content helps keep things much cleaner and more modular in the workflow xml file.

EXERCISE: MAKING SURE YOU'RE IN THE RIGHT PLACE

➤ Let's start by importing some content created for our email notifications.

1. **Open** the *Menu*.
2. **Go to** the Site Administration Panel and make sure you are on the S.P.A.C.E. Site.
3. **Go to** *Content→Web Content*.
4. **Click** on the *Options* button at the top-right corner.

EXERCISE: IMPORTING STYLED WEB CONTENT

1. **Choose** *Export/Import*.
2. **Click** the *Import Tab*.
3. **Choose** the *email-styled-banner.lar* from *exercises/front-end-developer-exercises/07-workflow-templates*.
4. **Click** *Continue→Import*.
5. **Close** the pop-up.

EXERCISE: ADDING WEB CONTENT TO THE WORKFLOW

- ❖ Now we have a simple example of a S.P.A.C.E. banner content article that can be referenced in our notification template.
- ❖ Let's update our email notification.
 1. **Open** the *o2-workflow-web-content* snippet.
 2. **Copy** the contents of the snippet.
 3. **Replace** *<!-- Add snippet o2-workflow-web-content here -->*, on line 65 of the *assignment-grading.xml* file with the contents of the snippet.
 4. **Save** the file.
 - ❖ Here we're using the *serviceLocator* variable to reference a web content article by its Site ID and title.
 - ❖ Next, let's update the article parameters.

EXERCISE: SITE ID AND ARTICLE NAME

1. **Open** the *Menu* in Liferay.
2. **Go to** *Configuration*→*Site Settings* in the Site Administration Panel.
3. **Copy** the Site ID found at the top.
4. **Replace** the first parameter in
journalArticleLocalService.getArticleByUrlTitle(o, "header-web-content")
on line 67 of the *assignment-grading.xml* file with the Site ID.
5. **Replace** the "header-web-content" with "email-styled-banner".
6. **Save** the file.
- ✓ Now, our email template is using a web content article to provide a header banner in our emails.

THE SERVICELOCATOR

- Our snippet is referencing web content by using the following:

```
<#assign journalArticleLocalService = serviceLocator.findService("com.liferay.journal.service.JournalArticleLocalService") />
```
- Because the serviceLocator can get access to the database, it is restricted by default.
- This variable can be unlocked by removing it from the Freemarker Engine Restricted Variables found in *Menu→Control Panel→Configuration→System Settings→Foundation*.
 - This is typically done by a System Administrator.



MAKING THE WORKFLOW DEFINITION AVAILABLE

- Once the notification templates have been added, the xml can be handed off to an administrator.
- Administrators can make the workflow definition xml available for use by doing the following:
 - Going to *Menu→Control Panel→Configuration→Workflow Definition*
 - Uploading the xml definition
- Once the definition is available, workflow can be configured on document folders where Students upload their assignments.
- When an assignment is uploaded, Instructors can grade and provide comments.
- Our notification template will take those comments and provide the feedback directly to the students via email.

Notes:

Chapter 8

Creating Different Front Ends for Applications



CUSTOMIZING APPLICATION DISPLAYS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

KEEPING THE LOOK

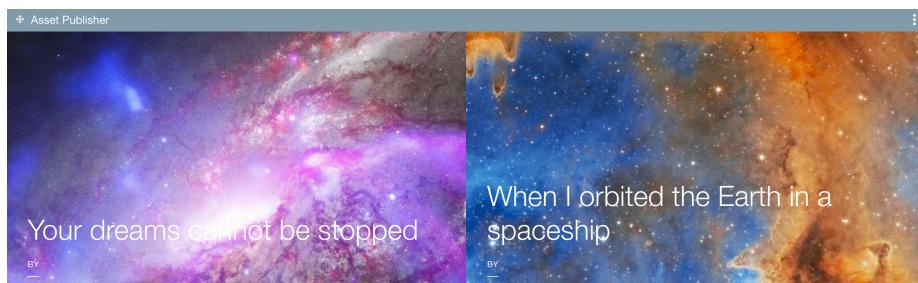
- We want to keep applying our look and feel throughout Liferay.
- Templates give us the tools to create consistent styling in most places:
 - *Theme Templates* control the basic HTML and page styling for every page in Liferay
 - *Layout Templates* control application layout on pages
 - *Web Content Templates* control the look and feel and functionality of Web Content
 - *Workflow Notification Templates* apply our styles to Workflow notifications
- That takes care of most of the areas of Liferay we can directly control, aside from additional features.
- Many features are implemented in applications.
- Ideally, we want to make sure applications in Liferay look, feel, and behave consistent with our branding.

CUSTOMIZING APPLICATIONS

- Usually applications are built by developers using Java and complex back-end tooling.
- Customizing the display usually means becoming familiar with:
 - Java Web App structure
 - JSPs
 - Taglibs in JSPs
 - JSF
- In addition, you'd also need to have an environment set up to modify, build, and deploy applications in Liferay.
- Fortunately, Liferay lets us *override the default display* of lots of applications.
- Just like Web Content Display, applications can also support Templates.
- *Application Display Templates* take the same templating framework, and apply it to applications.

WHAT ARE APPLICATION DISPLAY TEMPLATES?

- The *Application Display Template (ADT)* framework allows administrators to override the default display of supported applications, removing limitations to the way your applications display content.
- With ADTs, you can define custom templates used to render asset-centric applications.



APPLICATIONS INCLUDED IN THE ADT FRAMEWORK?

- Any application can support ADTs, including custom-built applications.
- Liferay provides lots of content applications with full ADT support.
- The following applications support ADTs:
 - Asset Categories Navigation
 - Asset Publisher
 - Asset Tags Navigation
 - Blogs
 - Breadcrumb
 - Language
 - Media Gallery
 - Navigation Menu
 - RSS
 - SiteMap
 - Wiki

WHY USE ADTS?

- ADTs can create different displays for applications without back-end development.
- Use the same knowledge for building Web Content Templates, Workflow Notification Templates, and ADTs.
- ADTs provide the ability to:
 1. Create multiple views for a single application that can be selected by a Site Administrator
 2. Develop with advanced back-end functionality like serviceLocator and tag libraries through the GUI without having to deploy your changes to the server
 3. Use advanced styling options like Lexicon CSS

Notes:



STYLING APPLICATIONS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

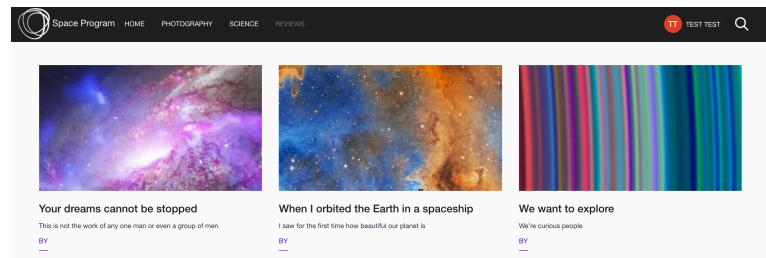
No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT YOU NEED

- This module uses:
 - Installed theme:
 - `space-program-theme.war`
- You can find the theme in *exercises/front-end-developer-exercises/05-front-end-developer/03-theme-development/solutions*.
- Our Theme includes some content and *Application Display Templates (ADTs)* for us to take a look at.

ASSET PUBLISHER ADT EXAMPLE

- ❖ Let's look at an example of an ADT bundled in our theme.
 - The theme let us create some useful assets in a sample site
- ❖ We have a number of Asset Publisher ADTs in use on the sample *Space Program* site generated by the Resources Importer.
- ❖ We'll look at the *Entry List 3 Item* ADT.



WWW.LIFERAY.COM

LIFERAY

ADTS AND WEB CONTENT TEMPLATES

- ❖ *Web Content Templates* and ADTs use the same basic type of template.
- ❖ Since they both use FreeMarker, it's easy to write for both.
- ❖ The difficulty can be in remembering what makes them different.
- ❖ ADTs reinforce styling, but have the main purpose of displaying (*rendering*) different types of assets.
- ❖ Let's look at our example ADT in detail so you can see what that looks like.
- ❖ This ADT displays content articles as image tiles with information below with 3 columns.

WWW.LIFERAY.COM

265

LIFERAY

EXERCISE: NAVIGATING TO ADT CREATION UI

- ❖ You'll want to make sure you're on the *Space Program* site to check out these examples.

1. **Open** the *Menu*.
2. **Go to** *Site Administration*→*Configuration*→*Application Display Templates* in the Space Program Site.
3. **Open** the *Entry List 3 items* ADT.

```
Name: Entry List 3 items ADT  
Fields:  
Asset Entries *  
Asset Entry  
Script:  
1 <@if entriesHasContent:  
2   ${request.setAttribute("displayMode", 2)}  
3   ${request.setAttribute("col1M", "col-md-4")}  
4   <div class="blog-list container-fluid-1280">  
5     <div class="row">  
6       <@for entries as curEntries:  
7         <@sign  
8           <@assign  
9             osmosisRenderer = curEntry.getAssetItemRender()  
10            viewURL = (osmosisRenderer.getBehavior() != "showfullContent")?then(assetPub  
listHelper.getAssetViewURL(renderRequest, renderResponse, curEntry, true),  
assetPublisherHelper.getAssetViewURL(renderRequest, renderResponse, curEntry))
```

ADTS HAVE ONLY ONE JOB

- ❖ The main job of ADTs is to *display dynamic data in markup*.
- ❖ This ADT is for the *Asset Publisher*, whose main content is *assets*.
- ❖ You'll notice the ADT gets a list of assets to display.
- ❖ Let's look more closely at how the assets are displayed and styled consistently.

MAKING ADT DESIGN EASIER

- ❖ The built-in editor makes it easier to reference variables.
- ❖ You can see an example of that in the *Entry List 3 Item* ADT.
- ❖ The ADT takes the default Asset Entries * field and expands upon it.
- ❖ Here is what is generated when you select Asset Entries * in the editor:

```
<#if entries?has_content>
    <#list entries as curEntry>
        ${curEntry.getTitle(locale)}
    </#list>
</#if>
```

ADDING CUSTOM ADT CODE

- ❖ Within the `<#if>` Structure, there are attributes set on the displayed content:

```
<#if entries?has_content>
    ${request.setAttribute("displayMode", 2)}
    ${request.setAttribute("colMd", "col-md-4")}
    ...
    ${request.setAttribute("author", " ")}
    ${request.setAttribute("displayMode", 0)}
    ${request.setAttribute("viewURL", " ")}
</#if>
```

- ❖ We also have this markup added:

```
<div class="blog-list container-fluid-1280">
    <div class="row">
        <#list entries as curEntry>
        ...
        </#list>
    </div>
</div>
```

EXERCISE: CREATING A NEWS FEED ADT

➤ Now that we've seen an ADT breakdown, let's add our own ADT for the S.P.A.C.E. front page *News and Events*.

1. **Open** the *Menu*.
2. **Click** on the *Site Selector* in *Site Administration*.
3. **Choose** the S.P.A.C.E. Site.
4. **Go to** *Configuration*→*Application Display Templates*.
5. **Click** *Add*→*Asset Publisher Template*.

EXERCISE: NAMING THE ADT

1. **Type** to name the ADT *Card List - Feed*.
2. **Open** the *card-list-feed-ap-adt.ftl* from *exercises/front-end-developer-exercises/o8-application-display-templates*.
3. **Copy** the contents.
4. **Paste** the contents in the template editor.
5. **Click** *Save*.

CARD LIST - FEED: EXPLANATION

1. In the Card List – Feed ADT, we check to see if there are any entries, and then iterate through the list.
 - You can hover over Asset Entries * for more information on entries.
2. Next, we use Lexicon CSS Cards to create the card layout.
 - <http://liferay.github.io/clay/content/cards/>
3. Finally, we'll display the article title, URL, and summary from our Asset Renderer.
 - As we're displaying Web Content Articles, see JournalArticleAssetRenderer for more information.
4. As front-end developers, we can collaborate with our back-end team to display dynamic content.

ASSET URLs

- We're using a URL for the article called *View in Context*.
- *View in Context* needs the article to be displayed on a page in order to work.
- If you'd like to test the URL, you'll want to add the Web Content Article to a page somewhere before using the link.

EXERCISE: CREATING AN EVENT FEED ADT

1. **Click** *Add→Asset Publisher Template*.
2. **Type** to name the ADT *Card List - Events*.
3. **Open** the *card-list-events-ap-adt.ftl* from *exercises/front-end-developer-exercises/o8-application-display-templates*.
4. **Copy** the contents.
5. **Paste** the contents into the template editor.
6. **Click** *Save*.

CARD LIST - EVENTS: EXPLANATION

1. In the *Card List - Events* ADT, the code is almost identical except for a few CSS changes.
 - » Instead of an rss icon, we displayed a calendar icon and changed the font.
2. The main difference is that we configured *Asset Publisher* to pull *Web Content Articles* that use the *Events Web Content Structure*.
3. We'll see the results of these changes in a few slides.
4. *Asset Publisher*, *Application Display Templates*, and *Web Content Templates* gives us precise control to create consistent branding and work better with our content marketing team in a way that allows both teams to focus on our areas of our expertise.

EXERCISE: CONFIGURING ASSET PUBLISHERS TO USE ADTS

- ❖ Once we have created our ADTs, we can use them with our Asset Publishers.
 1. **Click** *Go to Site* under *Site Administration* in the *Menu*.
 2. **Click** on the *Add* menu near the top right.
 3. **Open** the *Applications*→*Highlighted* section.
 - The Asset Publisher can also be found in the *Applications*→*Content Management* section.
 4. **Add** an Asset Publisher above the *About S.P.A.C.E.* article.
 5. **Add** an Asset Publisher below the *About S.P.A.C.E.* article.
- ❖ The Asset Publishers can be configured to display Web Content Structure, allowing our content team to make sure our News and Events are displaying on the front page.

EXERCISE: CONFIGURING THE FEED ADT

1. **Open** the *Options* menu of the first Asset Publisher.
2. **Click** *Configuration*.
3. **Click** the *Display Settings* tab.
4. **Choose** *Card List - Feed* under *Display Template*.
5. **Save** the changes.
6. **Close** the *Configuration* pop-up.

EXERCISE: CONFIGURING THE EVENTS TEMPLATE

1. **Open** the *Options* menu of the second Asset Publisher.
2. **Click** Configuration.
3. **Click** the *Display Settings* tab.
4. **Choose** *Card List - Events* under *Display Template*.
5. **Save** the changes.
6. **Close** the *Configuration* pop-up.

BRINGING IT ALL TOGETHER

- ❖ Let's import some content created by our content team in order to see everything in action.
- ❖ The content team has already configured the Asset Publishers to pull in the content they want to display.
- ❖ For the Asset Publisher at the top, they've configured it to pull in Web Content Articles that use the *News Structure*.
- ❖ For the second Asset Publisher, they've configured it to pull in Web Content Articles that use the *Events Structure*.
- ❖ The front-end team provided consistent styling for the Web Content using Web Content Templates.
- ❖ The front-end team will be responsible for applying the design, which we've completed above with ADTs.

EXERCISE: IMPORTING CONTENT USING ADTS

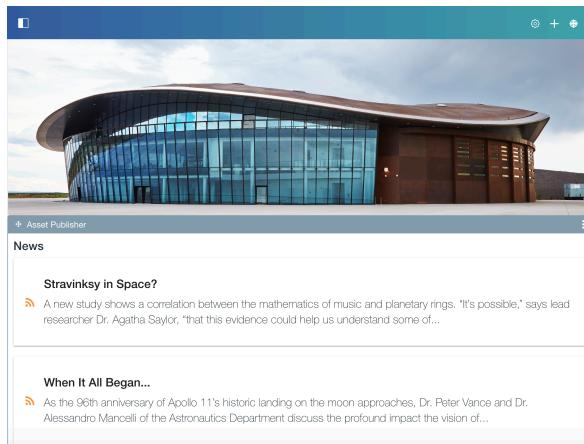
1. **Go to** *Menu→Site Administration→Publishing.*
 2. **Click** on *Import*.
 3. **Click** on the *Add* button at the bottom right.
 4. **Choose** the *Landing_Page_Redesign.lar* from *exercises/front-end-developer-exercises/o8-application-display-templates/space-front-page-content*.
 5. **Click** *Continue→Import*.
- ✓ Now we have content on display with our templates!

EXERCISE: APPLYING THE THEME

- ❖ When we imported our content changes, it reverted the theme selection.
 - ❖ Let's go ahead and re-apply the Space Program theme.
1. **Go to** *Menu→Site Administration→Navigation.*
 2. **Open** the *Options* menu next to *Public Pages*.
 3. **Click** *Configure*.
 4. **Click** *Change Current Theme*.
 5. **Choose** the *Space Program Theme*
 6. **Click** *Save*.
- ✓ Now our theme is applied!

READY FOR LIFT OFF!

- Now we've changed the overall look-and-feel of the S.P.A.C.E. site and its content.



WWW.LIFERAY.COM

LIFERAY.

Notes:

Appendix - Lexicon Details



INTRODUCING LEXICON CSS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

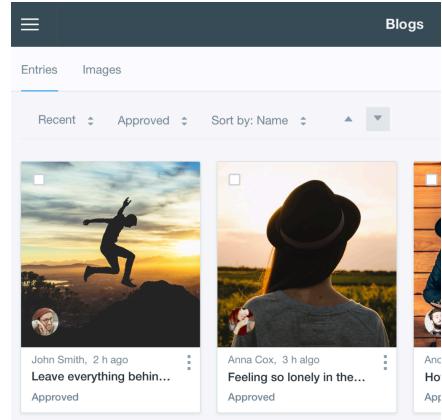
WHAT IS LEXICON CSS?

- ❖ Lexicon CSS is a web implementation of the Liferay's Lexicon Design Language.
- ❖ Lexicon CSS provides style guidelines and best practices for designing web applications.
- ❖ It is meant to be fluid and extensible while providing a complete collection of visual and interactive patterns.
- ❖ It is an extension of the Bootstrap 3 Framework and is built with Sass.
- ❖ Lexicon CSS fills the front-end gaps between Bootstrap and the specific needs of Liferay.
- ❖ Lexicon CSS is optimized for the most common operations and takes the best from the most popular modern UI patterns available.
- ❖ These patterns can be carefully combined, like Lego pieces, to obtain the desired user experience.



LEXICON CSS FEATURES

- Lexicon CSS has added components and features to cover more use cases:
 - Aspect Ratio
 - Cards
 - Dropdown Wide and Dropdown Full
 - Figures
 - Nameplates
 - Sidebar/Sidenav
 - Stickers
 - SVG Icons
 - Timelines
 - Toggles



REUSABLE PATTERNS

- We've also added a lot of reusable CSS patterns to help accomplish time-consuming tasks such as:
 - Truncating text
 - Content filling the remaining container width
 - Truncating text inside table cells
 - Table cells filling remaining container width and table cells only being as wide as their contents
 - Open and closed icons inside collapsible panels
 - Nested vertical navigations
 - Slideout panels
 - Notification icons/messages
 - Vertical alignment of content

NEW LEXICON CSS ICONS

- In previous versions of Liferay, the Bootstrap implementation in AUI took advantage of Font Awesome and Glyphicons for scalable icons.
- Lexicon CSS adds its own set of SVG icons that are bundled in the base theme.
- SVG gives developers a greater amount of freedom in styling the icons, as well as a higher level of fidelity and clarity in the icons.
- The SVG icons also allow for multi-color styling.

☒ add-cell	☒ add-column	☒ add-row
● adjust	☰ align-center	☰ align-justify
☰ align-left	☰ align-right	▽ angle-down
< angle-left	> angle-right	^K angle-up
✉ archive	★ asterisk	♫ audio
☰ bars	✉ blogs	:bold
☒ bookmarks	📅 calendar	📷 camera
☰ cards	☒ cards2	▼ caret-bottom
♦ caret-double-l	♦ caret-double	▲ caret-top

USING FONT AWESOME

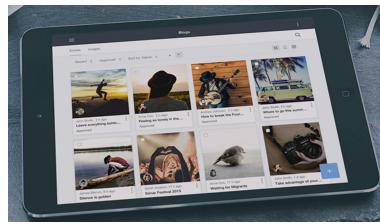
- If you've used Font Awesome in the past, you can continue to use it in your theme by adding the following in your `_aui_variables.scss` file:

```
// Icon paths
$FontAwesomePath: "aui/lexicon/fonts/alloy-font-awesome/font";
$font-awesome-path: "aui/lexicon/fonts/alloy-font-awesome/font";
$icon-font-path: "aui/lexicon/fonts/";
```

☰ align-center	☰ align-justify	☰ align-left	☰ align-right
:bold	">% chain (alias)	✂ chain-broken	📋 clipboard
☰ columns	✉ copy (alias)	✂ cut (alias)	☰ dedent (alias)
eraser	📄 file	□ file-o	📄 file-text
📄 file-text-o	📄 files-o	disk floppy-o	A font
H header	☰ indent	I italic	% link
☰ list	☰ list-alt	☰ list-ol	☰ list-ul

LEXICON CSS RESPONSIVE DESIGN

- Lexicon CSS has replaced the previous `respond-to` mixin with explicit media queries.
- Now, to include custom responsive design, you can use the following:
 - **Phone:** `include media-query(null, $screen-xs-max)`
 - **Tablet:** `include media-query(sm, $screen-sm-max)`
 - **Phone - Tablet:** `include media-query(null, $breakpoint_tablet - 1)`
 - **Tablet - Desktop:** `include sm`
 - **Desktop:** `include media-query($breakpoint_tablet, null)`



WWW.LIFERAY.COM

 LIFERAY.

LEXICON BASE

- When building a theme, developers start with the *Lexicon Base* theme.
- *Lexicon Base* is our Bootstrap API extension and includes all of the default Lexicon CSS styles.
- When a theme is created, developers can choose from the following options as a starting point for customization:
 - Building on the *Unstyled* base:
 - This base theme does not include the *Lexicon Base*, allowing developers to have full control over styling for every aspect of the platform.
 - Building on the *Styled* base:
 - This base theme includes the *Lexicon Base* theme.
 - It adds all the features and components we need and inherits Bootstrap's styles.

WWW.LIFERAY.COM

ATLAS

- ❖ Though *Lexicon Base* is the default import of a *Styled* theme, developers can also easily use what is called the *Atlas* theme.
- ❖ *Atlas* is Liferay's custom Bootstrap theme, used in the Classic Theme.
- ❖ Its purpose is to overwrite and manipulate Bootstrap and Lexicon Base to create our Classic look-and-feel.
- ❖ The benefit to using *Atlas* is that you'll gain some classic styles without trying to customize the existing Classic Theme itself.

USING ATLAS IN YOUR CUSTOM THEME

- ❖ Using *Atlas* instead of *Lexicon Base* is a simple switch in imports.
- ❖ To start, you can import *Atlas* in an `aui.scss` file:
`@import "aui/lexicon/atlas";`
- ❖ Then you can import the various packages, variables, and mixins for *Atlas* in an `_imports.scss` file:
`@import "bourbon";
@import "mixins";
@import "aui/lexicon/atlas-variables";
@import "aui/lexicon/bootstrap/mixins";
@import "aui/lexicon/lexicon-base/mixins";
@import "aui/lexicon/atlas-theme/mixins";`
- ❖ Building on the *Styled* base is recommended for theme development as it will contain both *Lexicon Base* and *Atlas*.

CUSTOMIZING THE CLASSIC THEME

- ❖ If you would like to include all of the Classic Theme instead of just the base styles of *Atlas*, you can copy over the files located in:

```
frontend-theme-classic/src/css  
frontend-theme-classic/src/images  
frontend-theme-classic/src/js  
frontend-theme-classic/src/templates
```

- ❖ This can also be accomplished using the Liferay Theme Tasks module `gulp kickstart` command and following the prompts.

```
[09:26:53] Starting 'kickstart'...
[09:26:53] Warning: the kickstart task will potentially overwrite files in your
src directory
? Where would you like to search?
  Search globally installed npm modules
  Search npm registry (published modules)
> Classic
```

USING LEXICON CSS

- ❖ Lexicon CSS can be used in both theme development and template development.
- ❖ Later, we'll create our own custom themes and templates that take advantage of these features.
- ❖ For a list of components, elements, etc., visit:
<http://liferay.github.io/clay/>



Notes:

Appendix - Modern Application Design with Templates



METAL.JS: SOY TEMPLATES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT ARE SOY TEMPLATES?

- ❖ *Soy Templates*, also known as Closure Templates, are developed by Google. They are a client-side and server-side templating system that helps you dynamically build reusable HTML and UI elements.
- ❖ They have a simple syntax, and you can customize them to fit your application's needs.
- ❖ In contrast to traditional templating systems, in which you must create one monolithic template per page, you can think of Soy Templates as small components that you compose to form your UI.
- ❖ You can also use the built-in message support to easily localize your applications.
- ❖ Metal.js is designed to work seamlessly with Soy Templates.

HOW IS THIS DIFFERENT?

- ❖ *Soy Templates* can be combined to build dynamic displays.
- ❖ This means we can build interface components and views piece by piece.
- ❖ Instead of having one big template for each “page” (view), we can break it out into components that can be reused:
 - Search bar
 - Table view
 - Data list
 - Paginator
 - Carousel
- ❖ Let’s see what this looks like.

MAKING JAVA PRETTY

- ❖ A classic use case for templates is to build pretty views for an application.
- ❖ Java can use tools like FreeMarker to create views from templates.
- ❖ The Java application provides data to the template and compiles it into a final output that will become the HTML for the page.

FROM WHOLE TO PARTS

- ❖ In this approach, we have to construct the entire view in one template.
- ❖ To break it down for ease of development and flexibility, we might be able to include other templates.
- ❖ Including external templates is limited, and doesn't allow for dynamically modifying the component based on environment or data.
- ❖ Soy Templates are components, and can be run independently.
- ❖ Building one view might mean calling multiple Soy Templates to build a search bar, buttons, and table view.
- ❖ A Soy Template can be *called* – just like a JavaScript function.

SOY TEMPLATE EXAMPLE

- ❖ A Soy Template can be self-contained:

```
{namespace Modal}

/**
 * This renders the component's whole content.
 * Note: has to be called ".render".
 */
{template .render}
    <div>Hello World</div>
{/template}
```

SOY TEMPLATE COMPONENTS

- Or contain other Soy Template components:

```
{namespace Modal}

/**
 * This renders the component's whole content.
 * Note: has to be called ".render".
 */
{template .render}
  {call .header /}
  <div>Hello World</div>
  {call .button}
    {param label: "Okay"}
  {/call}
{/template}
```

WHY USE SOY TEMPLATES?

- **Convenience:** Soy Templates provide an easy way to build pieces of HTML for your application's UI and help you separate application logic from display.
- **Language-neutral:** Soy Templates work with JavaScript or Java. You can write one template and share it between your client-side and server-side code.
- **Client-side speed:** Soy Templates are compiled to efficiently run JavaScript functions for maximum client-side performance.
- **Easy to read:** You can clearly see the structure of the output HTML from the structure of the template code. Messages for translation are inline for extra readability.

BENEFITS OF USING SOY TEMPLATES

- ❖ **Designed for programmers:** Soy Templates are simply functions that can call each other. The syntax includes constructs familiar to programmers. You can put multiple templates in one source file.
- ❖ **A tool, not a framework:** Soy Templates work well in any web application environment in conjunction with any libraries, frameworks, or other tools.
- ❖ **Battle-tested:** Soy Templates are used extensively in some of the largest web applications in the world, including Gmail and Google Docs.
- ❖ **Secure:** Soy Templates are contextually autoescaped to reduce the risk of XSS.

RELATIONSHIP BETWEEN METAL.JS AND SOY TEMPLATES

- ❖ For the integration between Metal.js and Soy Templates to work, the Soy Template files need to be compiled via one of our available build tools.
- ❖ That's because they don't just compile the code, but also add information that helps with the integration (like export declarations).
- ❖ The available build tools that correctly compile Soy Templates for Metal.js are:
 - **Metal Tasks:** <http://npmjs.com/package/gulp-metal> (already included when creating project via generator-metal)
 - **metal-cli:** <http://npmjs.com/package/metal-cli>
 - **metal-tools-soy:** <http://npmjs.com/package/metal-tools-soy>

REGISTER SOY TEMPLATES FROM METAL.JS

- ❖ The only thing you need to do to use Soy Templates in your Metal.js component is to call `Soy.register`, passing it your component class and the Soy Templates you're going to use, like this:

```
import templates from './MyComponent.soy';
import Component from 'metal-component';
import Soy from 'metal-soy';

class MyComponent extends Component {
}
Soy.register(MyComponent, templates);

export default MyComponent;
```

CASE FOR SOY TEMPLATES

- ❖ Soy Templates provide reusable components to build your displays.
- ❖ Simple syntax is easy to learn.
- ❖ Advanced features like calling functions, manipulating data, and localizing messages are built in.
- ❖ Soy Templates is the recommended templating language for Metal.js.

Notes:



METAL.JS: JSX

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS JSX?

- ❖ JSX is an interesting alternative to templates developed by the team at Facebook.
- ❖ It stands for JavaScript XML, and follows XML-type element declaration for web component development.
- ❖ It is an inline markup that looks like HTML and gets transformed into JavaScript.
- ❖ A JSX expression starts with an HTML-like open tag, and ends with the corresponding closing tag.
- ❖ While Soy Templates is the recommended templating system for Metal.js, JSX is a widely-used technology you may already be familiar with.
- ❖ If you already use JSX, or know how to use it, you can reduce some of your development time.

JSX EXAMPLE CODE

- ❖ Some example JSX code:

```
import JSXComponent from 'metal-jsx';

class Modal extends JSXComponent {
    render() {
        return <header class="modal-header">
            <button type="button" class="close">
                <span>x</span>
            </button>
            <h4>{this.config.header}</h4>
        </header>;
    }
}

export default Modal;
```

WHY USE JSX?

- ❖ **Familiarity:** Developers are familiar with XML, and JSX provides a similar type of element declaration.
- ❖ **Semantics:** JSX is easier to understand, as it follows a declarative type of programming.
- ❖ **Ease of development:** JSX provides a clean way to encapsulate all the logic and markup in one definition.

RELATIONSHIP BETWEEN METAL.JS AND JSX

- ❖ The only thing you need to do to use JSX in your Metal.js component is to extend from **JSXComponent**, like this:

```
import JSXComponent from 'metal-jsx';

class MyComponent extends JSXComponent {
}

export default MyComponent;
```

JSX RENDER AFTER EXTENDING

- ❖ Now that we've extended from **JSXComponent**, we can use JSX in the `render` method to specify what our component should render.

```
import JSXComponent from 'metal-jsx';

class MyComponent extends JSXComponent {
    render() {
        return <div class={this.config.cssClass}>
            Hello {this.name}
        </div>;
    }
}

MyComponent.STATE = {
    name: {
        validator: core.isString,
        value: 'World'
    }
};

export default MyComponent;
```

RENDERING JSX

- ❖ JSX components can either be rendered in the usual way (as seen here: <http://metaljs.com/docs/rendering-components.html>), or via the `JSXComponent.render` function, like this:

```
JSXComponent.render(MyComponent, {name: 'my-component'}, parent);
```

JSX COMPILATION

- ❖ For the integration between Metal.js and JSX to work, the JSX code needs to be compiled via a babel plugin called `babel-plugin-incremental-dom`.
- ❖ Using it directly means you'd need to configure it manually, so the Metal.js team also provides a *babel preset* that you can use instead: <https://www.npmjs.com/package/babel-preset-metal-jsx>.

Notes:



BASIC SOY TEMPLATE SYNTAX

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT ARE SOY TEMPLATES?

- ❖ We talked about Soy Templates earlier in the course, but let's review.
- ❖ Soy Templates, also known as Google Closure Templates, are a client-side and server-side templating system that helps you dynamically build reusable HTML and UI elements.
- ❖ The syntax is simple, and you can customize it to fit your application's needs.
- ❖ Soy Templates are implemented for both JavaScript and Java, so you can use the same templates on both the server-side and the client-side.
- ❖ This means you can render pages on the server before serving to the client, benefiting the user experience on first load.

HOW DO SOY TEMPLATES RELATE TO METAL.JS?

- ❖ Soy Templates are one of the template types compatible with Metal.js.
 - Soy Template support, along with other template options, is still in its initial steps and will be continually improved in the future.
- ❖ Using Soy Templates and Metal.js together is simple and makes it easy to create solid, lightweight, and flexible UI components.
- ❖ A Soy Portlet is a Liferay Portlet that uses Soy Templates and Metal.js as its front-end.

WHAT ARE THE BENEFITS OF SOY PORTLETS?

- ❖ Using Soy Templates as the template of your portlet gives you all the benefits of using Metal.js.
- ❖ You get the advantage of a framework that's built from the ground up with performance in mind.
- ❖ It's a versatile build system that you can leverage in a number of different ways.
- ❖ Your application would be written with future-ready *ECMAScript 2015* code, making it clean and easy to read.

WHAT DOES A SOY PORTLET LOOK LIKE?

- ❖ The file structure of a Soy Portlet would look similar to a regular application.
- ❖ Let's look at an example we'll call MySoyPortlet:

```
- my-soy-portlet
  - .lfrbuild-portal
  - build.gradle
  - bnd.bnd
  - package.json
  - src/main/
    - java/com/liferay/frontend/my/soy/portlet/web/internal/portlet
      - MySoyPortlet.java
    - resources/META-INF/resources/
      - MyComponent.es.js
      - MyComponent.soy
      - MyComponent.scss
```

EXTENDING THE SOYPORTLET CLASS

- ❖ MySoyPortlet can extend from the SoyPortlet class like this:

```
public class MySoyPortlet extends SoyPortlet {
    @Override
    public void render(RenderRequest renderRequest, RenderResponse
        renderResponse) {
        //do things here
    }
}
```

SOY TEMPLATE MAGIC

- ❖ And here is what the Soy Template magic looks like:

```
{namespace MyComponent}

/**
 * This renders the main content of the `MyComponent` component.
 * @param? content
 */
{template .render}
    <div class="my-component">
        <div class="my-component-content">
            <button type="button" class="close" data-onclick="hide">
                <span>×</span>
            </button>
            <h4>{$content ?: ''}</h4>
        </div>
    </div>
{/template}
```

METAL.JS JAVASCRIPT CODE WITH THE COMPONENT

- ❖ And the Metal.js JavaScript code to go along with this component:

```
'use strict';
import templates from './MyComponent.soy';
import Component from 'metal-component';
import Soy from 'metal-soy';
class MyComponent extends Component {
    hide() {
        // All Metal.js components already have a `visible` state which sets the
        // main element's `display` to "none" when set to false.
        this.visible = false;
    }
}
// This line is declaring that `MyComponent` will be using the given
// soy templates for
// rendering itself.
Soy.register(MyComponent, templates);

export default MyComponent;
```

SOY PORTLET EXAMPLES

- **Hello Soy Portlet:** A Hello World portlet built with Soy Templates:
 - <https://github.com/liferay/liferay-portal/tree/master/modules/apps/foundation/hello-soy/hello-soy-web>
- **Image Editor Portlet:** A portlet for editing images built with Soy Templates and Metal.js:
 - <https://github.com/liferay/liferay-portal/tree/master/modules/apps/foundation/frontend-image-editor/frontend-image-editor-web>

SOY TEMPLATE SYNTAX AND CONCEPTS

- The SoyPortlet provides the Soy Template file structure and Soy Template data.
- Let's take a closer look at the basic concepts and syntax.
 1. Template syntax such as:
 - Comments
 - Text
 2. Command
 3. Expression
 4. Functions

COMMENTS SYNTAX

- ❖ Comments in Soy Templates are similar to Java and JavaScript.
- ❖ If preceded by a white space, // begins a rest of line comment.
- ❖ Using /* */ delimits any text between /* and */ as a comment.

```
{template .soyComments}
  I Love Liferay<br> // What a great comment
  /* you can use
  multiline comments */
  // URL syntax is not interpreted as comment.
  http://www.google.com<br>
{/template}
```

TEXT SYNTAX

- ❖ Any character in a template that does not appear between braces {} is raw text.
- ❖ The compiler does not parse raw text, except for joining lines and removing indentation.
- ❖ This allows developers to create nicely formatted templates with proper indentation and readability.

```
{template ..soyTextSyntax}
  // These two lines will be joined by adding a space.
  First
  second.<br>
  // When either HTML or Soy tag border the join location the
  //lines will be joined without adding a space.
  <i>First</i>
  second.<br>
  First
  {''}second.<br>
{/template}
```

TEXT SYNTAX: SPECIAL CHARACTERS COMMANDS

- Soy Templates also provide a number of commands to generate raw text.
- This includes special character commands.
 - {sp}: A space
 - {nil}: An empty string
 - {\r}: Carriage return
 - {\n}: New line or line feed
 - {\t}: Tab
 - {lb}: Left brace
 - {rb}: Right brace

TEXT SYNTAX: COMMANDS

- There are a few other commands that can be used to generate text.
- The literal command allows you to include a block of raw text, as no processing happens between literal blocks and output is rendered as it appears in the template.

```
{template .soyLiteral}
  // Note: Lines are not joined and indentation is not stripped
  Literal : {literal}AA BB { CC DD } EE {sp}{\n}{rb} FF{/literal}
  </pre>
{/template}
```

- The print command will output the results of an expression:

```
{template .soyPrint}
  {print 'Boo!'}<br> // print a string
  {'Boo!'}<br> // the command name 'print' is implied
  {1 + 2}<br> // print the result of an expression
  {$boo}<br> // print a data value
  {1 + $two}<br> // print the result of an expression that uses a data value
{/template}
```

COMMANDS

- ❖ Beyond the special character and literal and print commands, there are many other commands available in Soy Templates.
- ❖ Commands are instructions provided to the compiler to create templates and add custom logic.
- ❖ Command syntax in Soy Templates is delimited by braces {}. The command name can be followed by additional tokens to form the command text.
- ❖ Let's look at a few common commands and their syntax.

COMMANDS: BASIC

- ❖ The {call} command can be used to call another template and return its output. You can also provide parameters to the call template.

```
{template .soyCaller}
  {call .soyCallee}
    {param firstName: $instructor.firstName /}
    {param lastName: $instructor.lastName /}
  {/call}
{/template}
{template .soyCallee}
  Hi $firstName$lastName, welcome to Liferay Training
{/template}
```

- ❖ Use the {let} command to define an intermediate value.

```
{let $isAbeforeB: $aaa < $bbb /}
```

COMMANDS: CONTROL FLOW

- ❖ The {if} commands can be used for conditional output.

```
{if $instructor.name == 'Nick'  
    <h1>Dude rocks</h1>  
{elseif $instructor.name == 'Jon'  
    <h1>Dude rocks more</h1>  
{else  
    <h1>We all rock</h1>  
{/if}}
```

- ❖ Similarly, the {switch} command can be used for conditional output.

```
{switch $instructor.name  
{case 'Nick'  
    <h1>Dude rocks</h1>  
{case 'Jon'  
    <h1>Dude rocks more</h1>  
{default  
    <h1>We all rock</h1>  
{/switch}}
```

COMMANDS: LOOPS

- ❖ The {foreach} command iterates a list.

```
{foreach $instructor in $instructors  
    <h1>{$instructor.name} rocks</h1>  
{ifempty  
    <h1>No teachers let's party</h1>  
{/foreach}}
```

- ❖ The {for} command is used for a numerical loop.

```
{for $i in range($instructors.size)}  
    $i little instructor went to the market.<br>  
{/for}
```

- ❖ Note: The range function can take one, two, or three arguments to allow the loop to control initial value, limit, and increment.

FUNCTIONS

- Beyond the `range()` function in the `{for}` command, there are a number of other functions available.
 - `isFirst(var)`, `isLast(var)`, and `index(var)` functions can be used with the `{foreach}` command.
 - `isNonnull(value)` function will return true if the given value is both defined and not the value null.
 - There are functions that operate on a list or map such as `length(list)`, `keys(map)`, `augmentMap(baseMap, additionalMap)`, and `quoteKeysIfJs(mapLiteral)`.

FUNCTION OPERATORS

- There are also functions that provide operations on numbers and text.
 - `round(number)` and `round(number, numDigitsAfterDecimalPoint)` will round to an integer or to a significant digit.
 - `floor(number)` returns the floor of the number.
 - `ceiling(number)` returns the ceiling of the number.
 - `min(number, number)` returns the min of the two numbers.
 - `max(number, number)` returns the max of the two numbers.
 - `randomInt(rangeArg)` A random integer between 0 and the rangeArg
 - `strContains(str, subStr)` checks whether a string contains a substring.

REFERENCE

- ❖ You can find more references and samples at:
 1. <https://developers.google.com/closure/templates/docs/commands>
 2. <https://github.com/google/closure-templates/blob/master/examples/features.soy>
 3. Liferay Source:
<liferay-portal/modules/apps/foundation/hello-soy/hello-soy-web/>

Notes:



BUILDING APPLICATION UIS WITH SOY TEMPLATES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

REQUIREMENTS

- S.P.A.C.E. would like to add a new application for students and instructors to manage an activity list.
- The development team has been tasked with creating this new application.
- The back-end developers have created a new set of APIs using Liferay Service Builder and have exposed JSON endpoints to these services.
- They have also decided to use a SoyPortlet to render this new application.
- As front-end-developers, we need to build a nice UI for this new application.

EXERCISE: API

- ❖ First, let's take a look at this new API and see what is available to us.
 - ❖ For this exercise, you'll need all the files available in your `appendix-soy-templates` directory.
1. **Go to** `exercises/front-end-developer-exercises/appendix-soy-templates`.
 2. **Copy** the `com.liferay.todo.api.jar` and the `com.liferay.todo.service.jar`.
 3. **Go to** your `[LIFERAY_HOME]` directory here:
`liferay/bundles/liferay-dxp-digital-enterprise-[version]`.
 4. **Paste** the files in the `/deploy` folder.
 5. **Go to** `http://localhost:8080` in your browser.
 6. **Sign in** with your user account.
 7. **Go to** `http://localhost:8080/api/jsonws`.

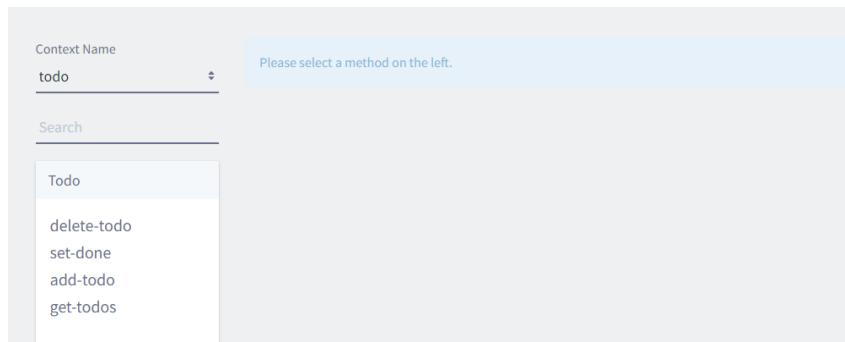
AVAILABLE JSON WEBSERVICES

- ❖ Your browser should now show the list of JSON WebServices available in the platform.
- ❖ The back-end developers have also notified us that the services are exposed under the `todo` Context name.



EXERCISE: CHANGING THE CONTEXT

1. Choose *todo* from the *Context Name* drop-down.
- ✓ You should now see a list of service endpoints such as *get-todos*, *add-todos*, *set-done*, and *delete-todos*.



The screenshot shows the Liferay Service Builder interface. On the left, there is a dropdown menu labeled "Context Name" with "todo" selected. Below it is a search bar. To the right, a message says "Please select a method on the left." A dropdown menu is open, listing the following service endpoints:

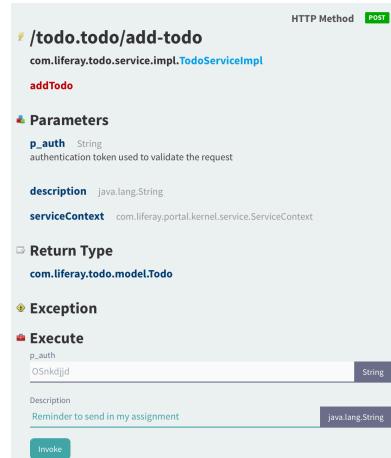
- Todo
- delete-todo
- set-done
- add-todo
- get-todos

WWW.LIFERAY.COM

LIFERAY

EXERCISE: SETTING A REMINDER

1. Choose *add-todo*.
2. Type *Reminder to send in my assignment* into the *Description* input field.
3. Click the *Invoke* button.



The screenshot shows the Liferay Service Builder interface for the */todo.todo/add-todo* endpoint. The HTTP Method is set to *POST*. The service class is *com.liferay.todo.service.impl.TodoServiceImpl* and the method is *addTodo*.
Parameters

- p_auth** String authentication token used to validate the request
- description** java.lang.String
- serviceContext** com.liferay.portal.kernel.service.ServiceContext

Return Type

- com.liferay.todo.model.Todo

Exception

- No exceptions defined.

Execute

- p_auth** OSnkljjd
- Description** Reminder to send in my assignment

Invoke button

WWW.LIFERAY.COM

LIFERAY

EXERCISE: USING JAVASCRIPT LATER

1. **Click** the JavaScript example tab to see an example of JavaScript invoking this service. We'll take note of this and use it in our code.
- ❖ *Bonus: You can play around with the other service endpoints to get a feel for what the API is offering.*



WWW.LIFERAY.COM

 LIFERAY

SOY TEMPLATES

- ❖ As mentioned previously, the back-end developers have already developed the SoyPortlet.
- ❖ The team has placed a list of todo into the Soy Template, which will allow us to access the data for display.

```
long groupId = themeDisplay.getScopeGroupId();
long userId = themeDisplay.getUserId();

List<Todo> todos = TodoLocalServiceUtil.getTodosByUserIdAndGroupId(userId,
groupId);

JSONSerializer jsonSerializer = JSONFactoryUtil.createJSONSerializer();
JSONDeserializer jsonDeserializer = JSONFactoryUtil.createJSONDeserializer();
List<Object> todoContainer = new ArrayList<>();
for (Todo todo : todos) {
    String json = jsonSerializer.serializeDeep(todo);
    todoContainer.add(jsonDeserializer.deserialize(json));
}
template.put("todos", todoContainer);
```

WWW.LIFERAY.COM

 LIFERAY

EXERCISE: SOY TEMPLATES

- ❖ As front-end developers, we are responsible for creating the following resources to render the UI:
 - **Todo.soy**: This file will be our primary Soy Template file.
 - **Todo.es.js**: This will be our Metal.js file.
 - **Todo.scss**: This will be our SCSS file to provide our styling.

 1. **Open** Brackets if it's not already open.
 2. **Click** on the dropdown in the left-hand side bar.
 3. **Click** Open Folder...
 4. **Go to** `exercises/front-end-developer-exercises/appendix-soy-templates`.
 5. **Choose** the snippets folder.

- ❖ Let's start by laying out our Soy Template to provide the UI elements for our application.

EXERCISE: INPUT FIELDS

1. **Go to** `exercises/front-end-developer-exercises/appendix-soy-templates/src`.
 2. **Open** the `Todo.soy` file in Brackets.
 3. **Copy** the contents of the `o1-input-fields` snippet.
 4. **Replace** `<%-- Insert O1-input-fields ---%>` in the `Todo.soy` file with the `o1-input-fields` snippet.
 - Format as needed.
 - Take note of the `data-onclick="addTodo"` attribute on our add button. We will provide the `onclick` method shortly.
-
- ❖ Next, we will create the UI elements to list the list of `Todos`.

EXERCISE: CREATING UI ELEMENTS

1. **Copy** the contents of the *o2-list-todo* snippet.
2. **Replace** <%-- Insert 02-list-todo ---%> in *Todo.soy* with the *o2-list-todo* snippet.
 - The snippet is taking advantage of the {foreach} and {if} commands to loop through the list of *todos* provided to the template and make UI decisions as to whether to mark the item as done.
 - We have also added another *onclick* method to the checkbox `data-onclick="finishTodo"`.

```
checked="true"  
data-item-is-done="true"
```

EXERCISE: VIEWING COMPLETED TASKS

- For the final piece of our UI, we will provide a toggle to allow the User to view completed *todo* tasks.
1. **Copy** the contents of the *o3-done-toggle* snippet.
 2. **Replace** <%-- Insert 03-done-toggle ---%> in *Todo.soy* with the *o3-done-toggle* snippet.
 3. **Save** the file.
 - Once again, we have attached an *onclick* method to our toggle input, `data-onclick="toggleTodo"`, which we will implement in a moment.
- Now that we have our UI elements laid out in our Soy Template, review the various CSS classes we've added onto our UI elements.
 - You can reference the Lexicon CSS modules for additional information about these CSS classes.

EXERCISE: METAL.JS

- With our UI now laid out in the Soy Template, let's provide the implementation of the *onclick* methods required by the UI.
- 1. **Open** the *Todo.es.js* file in *Brackets*.
 - Here, you will see that we have defined a *Todo* Metal.js component and registered our Soy Templates.

```
class Todo extends Component {  
    ...  
    // Register component  
    Soy.register(Todo, templates);  
    ...  
}
```

EXERCISE: ADD TODO

1. **Copy** the contents of the *o4-add-todo-method* snippet.
2. **Replace** <%-- Insert 04-add-todo-method ---%> in *Todo.es.js* with the *o4-add-todo-method* snippet.
 - Take a look at the code. You can see that we're using the same JavaScript code to invoke the service as we saw previously when we were browsing the JSON service endpoints.

```
Liferay.Service(  
    '/todo.todo/add-todo',  
    {  
        description: todoValue,  
    },  
    ...  
);
```

EXERCISE: ADDING THE OTHER METHODS

- Let's repeat this process and add the other two methods.
1. **Copy** the contents of the *o5-finishTodo-method* snippet.
 2. **Replace** <%-- Insert 05-finishtodo-method ---%> in *Todo.es.js* with the *o5-finishTodo-method* snippet.
 - Again, you can see that the code closely mirrors the JavaScript code from before.
 3. **Copy** the contents of the *o6-toggleTodo-method* snippet.
 4. **Replace** <%-- Insert 06-toggleTodo-method ---%> in *Todo.es.js* with the *o6-toggleTodo-method* snippet.
 5. **Save** the file.

EXERCISE: CSS FOR THE APPLICATION

- Now that we have our UI and its JavaScript methods created using Soy Templates and Metal.js components, let's add the finishing touches and provide some CSS for our application.
 - Then we can ship the completed code to the back-end developer and see our application in action.
1. **Open** the *Todo.scss* file in *Brackets*.
 2. **Insert** the *o7-todo-sass* snippet.
 3. **Go to** *exercises/front-end-developer-exercises/appendix-soy-templates*.
 4. **Copy** the *com.liferay.todo.web.jar*.
 5. **Go to** your *[LIFERAY_HOME]* directory here:
liferay/bundles/liferay-dxp-digital-enterprise-[version].
 6. **Paste** the JAR file into the */deploy* folder.

EXERCISE: VERIFYING THE APPLICATION

1. **Go to** <http://localhost:8080> in your browser.
 2. **Sign in** with your User account.
 3. **Open** the *Add* menu at the top right.
 4. **Click** to open *Applications→Tools*.
 5. **Drop** the *Todo Portlet* Application onto the page.
- ✓ Feel free to test the application's functionality by adding a new *Todo* and toggling the items.

VERIFYING THE APPLICATION

Todo Portlet

Here is the todo list

Add a todo

Buy some cheese

Reminder to send in my assignment

Show completed tasks

Notes:

Appendix - Customizing the Alloy Editor



ALLOY EDITOR IN LIFERAY

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS ALLOY EDITOR?

- Alloy Editor is a modern “What you see is what you get” editor built on top of CKEditor, designed to create modern and nicely-styled web content.

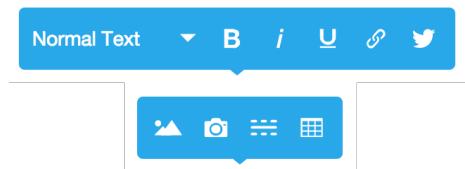
The screenshot shows the Alloy Editor interface. At the top, there's a navigation bar with links for "Live Demo", "What's New", "Guides", "Features", "API Docs", "Contributing", and "About". Below the navigation bar is a toolbar with icons for "Normal Text", "B", "i", "U", "P", and "V". The main content area contains text about AlloyEditor being a WYSIWYG Editor built on top of CKEditor by Liferay. It includes a note about editing content on the fly and provides examples of how styling appears when text is selected. A sidebar on the right lists helpful links: "Editing the content on the fly", "example image", "Paste content", and "Drag&Drop images".

WHAT MAKES ALLOY EDITOR DIFFERENT?

- ❖ Editing web content without using iFrame is not a new concept.
- ❖ However, the toolbar of most editors appears on top or on bottom of the container.
- ❖ Wouldn't it be better to have a modern UI with all the benefits of CKEditor?
- ❖ Alloy Editor provides a UI with context toolbars, which change their appearance depending on the selected element - image or text.

CUSTOMIZING THE TOOLBAR

- ❖ Developers can add or remove buttons to these toolbars on the fly.



- ❖ Alloy Editor also offers plugins that allow dragging and dropping of images directly in the editor or adding placeholders that work as the standard placeholder HTML5 attribute.

WHAT IS ALLOY EDITOR USED FOR?

- ❖ Alloy Editor is used by default in Liferay DXP.
- ❖ Applications that use the Alloy Editor include web content, blog posts, announcements, etc.
- ❖ Alloy Editor is flexible and configurable enough to do any job CKEditor can.

HOW TO ADD THE ALLOY EDITOR

- ❖ Creating an Alloy Editor instance is as easy as calling the Alloy Editor `editable` static method:
`AlloyEditor.editable('myContentEditable');`
- ❖ There's also another way.
- ❖ You can use the Liferay UI Input Editor taglib in your JSP:
`<liferay-ui:input-editor contents="<% content %>" editorName="alloyeditor" name="myAlloyEditor" placeholder="caption" showSource="<% false %>" />`
- ❖ Notice the `editorName` attribute is set to `alloyeditor`.

WHAT BROWSERS SUPPORT ALLOY EDITOR?

- ❖ AlloyEditor runs on the following browsers:
 - IE9+
 - Chrome
 - Firefox
 - Safari

Notes:



CONFIGURING AND CUSTOMIZING ALLOWED CONTENT RULES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

ALLOWING CONTENT IN THE EDITOR

- ❖ It's possible to configure the `allowedContent` of an editor through an OSGi module.
- ❖ For more information about the allowed content rules, please see the documentation on *CKEditor*:
http://docs.ckeditor.com#!/guide/dev_allowed_content_rules

EXTENDING THE EDITOR'S CONFIGURATION

- ❖ To modify the editor's configuration, you can create a module that has a component that implements the `EditorConfigContributor` interface.
- ❖ When you implement this interface, your module will provide a service that modifies the editors you'd like to change.

EXTENDING CONFIGURATION EXAMPLE

- ❖ As an example, you can create a generic OSGi module using your favorite third-party tool or use the *Blade CLI*.
- ❖ You can create a unique package name in the module's `src` directory and create a new Java class in that package.
- ❖ The class should extend the `BaseEditorConfigContributor` class.
- ❖ Directly above the class's declaration, you can insert a component annotation:

```
@Component(  
    property = {  
          
    },  
    service = EditorConfigContributor.class  
)
```

CHANGING THE ALLOWED CONTENT RULES

- For this example, you could change the allowedContent of the Web Content's AlloyEditor. The declaration would look like this:

```
import com.liferay.journal.constants.JournalPortletKeys;
@Component(
    property = {
        "editor.name=alloyeditor",
        "javax.portlet.name=" + JournalPortletKeys.JOURNAL,
        "service.ranking:Integer=100"
    },
    service = EditorConfigContributor.class
)
public class CustomJournalMediaEditorConfigContributor
    extends BaseEditorConfigContributor {
}
```

SPECIFYING CHANGES

- Next, changes need to be specified.
- Add the following method to the new class:

```
@Override
public void populateConfigJSONObject(
    JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
    ThemeDisplay themeDisplay,
    RequestBackedPortletURLFactory requestBackedPortletURLFactory) {
    ...
}
```

- In the populateConfigJSONObject method, the element value can be added for allowedContent, extraPlugins, and disallowedContent through the jsonObject parameter:

```
jsonObject.put("allowedContent", "p ");
jsonObject.put("disallowedContent", "h1 h2 h3 h4 h5 h6 ");
```

- It should look like the 01-configuration-example.java in the exercises folder.

FINISHING UP

- Now if we create a new web content article, the AlloyEditor will disallow all header tags except for <p> tags.

Notes:



CONFIGURING AND CUSTOMIZING THE TOOLBAR OPTIONS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

CUSTOMIZING TOOLBAR OPTIONS

- ❖ Liferay supports many different kinds of WYSIWYG editors that can be used in applications to edit content.
- ❖ Depending on the content you're editing, you may want to modify the editor to provide a better configuration for your needs.
- ❖ Let's look at how to extend the Liferay-supported WYSIWYG editor to add new or modify existing configurations.

EXTENDING THE EDITOR'S CONFIGURATION

- ❖ To modify the editor's configuration, you can create a module that has a component that implements the `EditorConfigContributor` interface.
- ❖ When you implement this interface, your module will provide a service that modifies the editors you'd like to change.
- ❖ A simple example of this is provided below.

EXTENDING TOOLBAR OPTIONS EXAMPLE

- ❖ You can create a generic OSGi module using your favorite third-party tool or using the *Blade CLI*.
- ❖ A Java class in a module's unique package should extend the `BaseEditorConfigContributor` class.
- ❖ Directly above the class's declaration, the following component annotation can be inserted:

```
@Component(  
    property = {  
        },  
    service = EditorConfigContributor.class  
)
```

ADDING VIDEO AND CAMERA BUTTONS

- For this example, we will see how to add the video and camera buttons to the Web Content's AlloyEditor. The declaration would look like this:

```
import com.liferay.journal.constants.JournalPortletKeys;

@Component(
    property = {
        "editor.name=alloyeditor",
        "javax.portlet.name=" + JournalPortletKeys.JOURNAL,
        "service.ranking:Integer=100"
    },
    service = EditorConfigContributor.class
)
public class CustomJournalMediaEditorConfigContributor
    extends BaseEditorConfigContributor {
}
```

SPECIFYING CHANGES

- The following method could be added to specify changes.

```
@Override
public void populateConfigJSONObject(
    JSONObject jsonObject, Map<String, Object> inputEditorTaglibAttributes,
    ThemeDisplay themeDisplay,
    RequestBackedPortletURLFactory requestBackedPortletURLFactory) {
}
```

- In the `populateConfigJSONObject` method, you need to instantiate a `JSONObject` that holds the current configuration of the editor. For instance, you could do something like this:

```
JSONObject toolbarsJSONObject = jsonObject.getJSONObject("toolbars");
```

MODIFYING CONFIGURATION

- ❖ With the `JSONObject` holding the editor's configuration, the configuration can be modified.
- ❖ Suppose you'd like to add a button to your editor's toolbar.
- ❖ To complete this, you'd need to extract the Add buttons out of your toolbar configuration object as a `JSONArray`, and then add the button to that `JSONArray`.

ADDING CAMERA CODE

- ❖ For example, the following code would add a Camera button to the editor's toolbar:

```
if (toolbarsJSONObject != null) {  
    JSONObject addJSONObject = toolbarsJSONObject.getJSONObject("add");  
  
    if (addJSONObject != null) {  
        JSONArray buttonsJSONArray = addJSONObject.getJSONArray("buttons");  
  
        buttonsJSONArray.put("camera");  
        buttonsJSONArray.put("video");  
    }  
  
    addJSONObject.put("buttons", buttonsJSONArray);  
  
    toolbarsJSONObject.put("add", addJSONObject);  
}
```

FINISHING UP

- ❖ All together it should look like the 02-toolbar-customization-example.java file in your exercises folder.
- ❖ Now if we create a new web content article, the AlloyEditor will have our new options.

Notes:



USING ALLOY EDITOR IN CUSTOM APPLICATIONS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

IMPLEMENTING ALLOY EDITOR INTO YOUR APPLICATION

- ❖ The AlloyEditor can easily be implemented into custom applications using the `<liferay-ui:input-editor />` taglib.
- ❖ Here is an example:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>

<div class="alloy-editor-container">
    <liferay-ui:input-editor contents="Default Content"
        cssClass="my-alloy-editor" editorName="alloyeditor"
        name="myAlloyEditor" placeholder="description" showSource="true" />
</div>
```

PASSING JAVASCRIPT FUNCTIONS

- ❖ You can pass JavaScript functions through `onBlurMethod`, `onChangeMethod`, `onFocusMethod`, and `onInitMethod`.

```
<%@ taglib prefix="aui" uri="http://liferay.com/tld/aui" %>
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
<div class="alloy-editor-container">
    <liferay-ui:input-editor contents="Default Content" cssClass="my-alloy-editor" editorName="alloyeditor" name="myAlloyEditor" onInitMethod="OnDescriptionEditorInit" placeholder="description" showSource="true" />
</div>
<aui:script>
    function <portlet:namespace />OnDescriptionEditorInit() {
        <c:if test="<%!= customAbstract %>">
            document.getElementById('<portlet:namespace />myAlloyEditor').setAttribute('contenteditable', false);
        </c:if>
    }
</aui:script>
```

ALLOY EDITOR ATTRIBUTES (I)

- ❖ As you can see, Liferay has made it easy to make use of the new AlloyEditor.
- ❖ Below is an overview of the attributes for the `liferay-ui:input-editor` taglib that are used with AlloyEditor:
- ❖ `<liferay-ui:input-editor />`
 - `autoCreate (java.lang.String)`: A string boolean to determine whether or not to show HTML edit view of editor initially
 - `contents (java.lang.String)`
 - `contentsLanguageId (java.lang.String)`: The ID of a language for the input editor's text
 - `cssClass (java.lang.String)`: A CSS class for styling the component
 - `data (java.util.Map)`: data that can be used as the editorConfig
 - `editorName (java.lang.String)`
 - `name (java.lang.String)`: A name for the input editor. The default value is `editor`.

ALLOY EDITOR ATTRIBUTES (II)

- <liferay-ui:input-editor />
 - onBlurMethod (java.lang.String): A function to be called when the input editor loses focus
 - onChangeMethod (java.lang.String): A function to be called on a change in the input editor
 - onFocusMethod (java.lang.String): A function to be called when the input editor gets focus
 - onInitMethod (java.lang.String): A function to be called when the input editor initializes
 - placeholder (java.lang.String): Placeholder text to display in the input editor
 - showSource (java.lang.String): Whether to enable editing the HTML source code of the content. The default value is *true*.
 - skipEditorLoading (boolean): Whether to skip loading resources necessary for the CKEditor. The default value is *false*.

Notes:



CREATING AND CONTRIBUTING NEW FUNCTIONALITY

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

ADDING FUNCTIONALITY

- It is possible to add additional AlloyEditor functionality through OSGi modules.
- Let's take a look at how to add a button to the editor.
- AlloyEditor is built on React.js and uses JSX to handle the interactions for each button in the editor.

ADDING THE OSGI MODULE

- Here is the folder structure of a module for adding a new button.

```
- frontend-editor-alloyeditor-accessibility-web
  - src
    - main
      - java
        - com/liferay/frontend/editor/alloyeditor/accessibility/web/
          - editor
            - configuration
              - AlloyEditorAccessibilityConfigContributor.java
          - servlet
            - taglib
              - AlloyEditorAccessibilityDynamicInclude.java
      - resources
      - META-INF
        - resources
        - js
          - button_image_alt.jsx
```

WHAT WE NEED TO COMPILE

- To compile our module, we need the following:
 1. .babelrc - needed since we are compiling JSX
 2. bnd.bnd

```
Bundle-Name: Liferay Frontend Editor AlloyEditor Accessibility Web
Bundle-SymbolicName: com.liferay.frontend.editor.alloyeditor.accessibility.web
accessibility.web
Bundle-Version: 1.0.2
Liferay-Releng-Module-Group-Description:
Liferay-Releng-Module-Group-Title: Rich Text Editors
Web-ContextPath:
/frontend-editor-alloyeditor-accessibility-web
```

BUILD.GRADLE

- ❖ Here is what the build.gradle looks like:

```
configJSModules {  
    enabled = false  
}  
dependencies {  
    provided group: "com.liferay.portal", name: "com.liferay.portal.kernel",  
    version: "2.0.0"  
    provided group: "javax.servlet", name: "javax.servlet-api", version: "3.0.  
    provided group: "org.osgi", name: "org.osgi.service.component.annotations"  
    version: "1.3.0"  
}  
transpileJS {  
    bundleFileName = "js/buttons.js"  
    globalName = "AlloyEditor.Buttons"  
    modules = "globals"  
    srcIncludes = "**/*.jsx"  
}
```

PACKAGE.JSON

- ❖ In the package.json, we've added the contents of some files since you'll need customizations in the build gradle.

```
{  
    "devDependencies": {  
        "babel-preset-react": "^6.11.1",  
        "metal-cli": "^2.0.0"  
    },  
    "name": "frontend-editor-alloyeditor-accessibility-web",  
    "version": "1.0.2"  
}
```

CONTENTS OF JSX FILE

- ❖ You can find the contents of the 03-button_image_alt.jsx file in the exercises folder.
- ❖ The file is well-documented, so take a look at the code.
- ❖ The important lines are those that reference the global AlloyEditor.
- ❖ You can create your own JavaScript functions for interactions with your button.

CONTENTS OF ACCESSIBILITYDYNAMICINCLUDE

- ❖ You can find the contents of our AlloyEditorAccessibilityDynamicInclude.java file in the exercises folder.
- ❖ This file adds our button to the global AlloyEditor that is included through the <liferay-util:dynamic-include /> taglib.
- ❖ This makes our button available to other AlloyEditor instances.

CONTENTS OF ACCESSIBILITYCONFIGCONTRIBUTOR

- ❖ You can find the contents of our `AlloyEditorAccessibilityConfigContributor.java` file in the `exercises` folder.
- ❖ This file handles where in the toolbar our new button should be.
- ❖ You can access the AlloyEditor toolbar and manipulate where the new button should be placed. Since there is no application name specified, this will be something added for the global AlloyEditor.

Notes:

Appendix - JavaScript



LIFERAY AMD MODULE LOADER

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS THE LIFERAY AMD MODULE LOADER?

- ❖ The Liferay AMD Module Loader is a JavaScript file and module loader.
- ❖ It can be found here:
<https://www.npmjs.com/package/liferay-amd-loader>

WHAT IS A JAVASCRIPT MODULE?

- ❖ JavaScript modules are a way to encapsulate a piece of code into a useful unit that exports its capability/value.
- ❖ This makes it easy for other modules to explicitly require this piece of code.
- ❖ Structuring an application this way makes it easier to see the broader scope, easier to find what you're looking for, and keeps things related.

WHAT'S THE PURPOSE OF A MODULE LOADER?

- ❖ A normal webpage usually loads JavaScript files via HTML script tags.
- ❖ That's fine for small websites, but when developing large-scale web applications, a better way to organize and load files is needed.
- ❖ A module loader allows an application to load dependencies easily by just specifying a string that identifies the module name.

HOW DO YOU DEFINE A MODULE?

- ❖ The Liferay AMD Module loader works with JavaScript modules that are in the AMD format. Here is a basic example of the definition of an AMD module:

```
define('aui-dialog', ['aui-node', 'aui-plugin-base'], function(node,
pluginBase) {
    return {
        log: function(text) {
            console.log('module aui-dialog: ' + text);
        }
    };
});
```

- ❖ You may specify that the module should be loaded on triggering some other module and only if some condition is being met.

CONDITIONAL LOADING

- ❖ This module should be loaded automatically if you request aui-test module, but only if some condition is being met.

```
define('aui-dialog', ['aui-node', 'aui-plugin-base'], function(node,
pluginBase) {
    return {
        log: function(text) {
            console.log('module aui-dialog: ' + text);
        }
    };
}, {
    condition: {
        trigger: 'aui-test',
        test: function() {
            var el = document.createElement('input');

            return ('placeholder' in el);
        }
    },
    path: 'aui-dialog.js'
});
```

HOW DO YOU LOAD A MODULE?

- ❖ Loading a module is as easy as passing the module name to the `require` method.

```
require('aui-dialog', function(base, test) {  
    // your code here  
}, function(error) {  
    console.error(error);  
});
```

MAPPING MODULE NAMES

- ❖ You can map module names to specific versions or other naming conventions.

```
--CONFIG__.maps = {  
    'liferay': 'liferay@1.0.0',  
    'liferay2': 'liferay@1.0.0'  
};
```

- ❖ Mapping a module will change its name in order to match the value, specified in the map.

```
require('liferay/html/js/autocomplete'...)
```

- ❖ Under the hood, this will be the same as:

```
require('liferay@1.0.0/html/js/autocomplete'...)
```

HOW IS THE LOADER USED IN LIFERAY 7?

- ❖ Tools, like the Liferay AMD Module Config Generator, have been integrated into the platform to make it easy for developers to create and load modules.
- ❖ An outline of the process is as follows:
 1. The tool scans your code and looks for amd modules `define(...)` statements.
 2. It will then name the module, if it is not named already.
 3. It takes note of that information, as well as the listed dependencies, and also any other configurations specified.

CONFIG.JSON

- ❖ Next, the tool creates a config.json file that may look something like this:

```
{  
    "frontend-js-web@1.0.0/html/js/parser": {  
        "dependencies": []  
    },  
    "frontend-js-web@1.0.0/html/js/list-display": {  
        "dependencies": ["exports"]  
    },  
    "frontend-js-web@1.0.0/html/js/autocomplete": {  
        "dependencies": ["exports", "./parser", "./list-display"]  
    }  
}
```

- ❖ You can see another example here: `modules/apps/foundation/frontend-js/frontend-js-metal-web/.task-cache/config.json`.
- ❖ This configuration object tells the loader which modules are available, where they are, and what dependencies they will require.

Notes:



JSON WEB SERVICES

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

JSON WEB SERVICES IN LIFERAY

- ❖ Liferay has many web services ready to use out of the box.
- ❖ These services include retrieving data and information about various assets, creating new assets, and even editing existing assets.
- ❖ To see a comprehensive list of the available web services, start up a bundle and navigate to <http://localhost:8080/api/jsonws>
- ❖ This list will include any custom web services that have been deployed to the bundle.
- ❖ These services are useful for creating single page applications, and can even be used to create custom front-ends both inside and outside of Liferay.

INVOKING WEB SERVICES VIA JAVASCRIPT

- ❖ In Liferay DXP, there is a global JavaScript object named `Liferay` that has many useful utilities.
- ❖ One method is `Liferay.Service`, which is used for invoking JSON web services.
- ❖ The `Service` method takes four possible arguments.

SERVICE METHOD ARGUMENTS

- ❖ **Required**

1. `service {string|object}`: Either the service name, or an object with the keys as the service to call, and the value as the service configuration object

- ❖ **Optional**

1. `data {object|node|string}`: The data to send to the service. If the object passed is the ID of a form or a form element, the form fields will be serialized and used as the data.
2. `successCallback {function}`: A function to execute when the server returns a response. It receives a JSON object as its first parameter.
3. `exceptionCallback {function}`: A function to execute when the response from the server contains a service exception. It receives the exception message as its first parameter.

SERVICE METHOD VS. STANDARD AJAX

- ❖ One of the major benefits of using the Service method vs. a standard ajax request is that it handles the authentication.

```
Liferay.Service(
  '/user/get-user-by-email-address',
  {
    companyId: Liferay.ThemeDisplay.getCompanyId(),
    emailAddress: 'test@liferay.com'
  },
  function(obj) {
    console.log(obj);
  }
);
```

- ❖ In this example, we are retrieving information about a user by passing in the `companyId` and `emailAddress` of the user in question.

RESPONSE DATA

- ❖ Response data resembles the following:

```
{
  "agreedToTermsOfUse": true,
  "comments": "",
  "companyId": "20116",
  "contactId": "20157",
  "createDate": 1471990639779,
  "defaultUser": false,
  "emailAddress": "test@liferay.com",
  "emailAddressVerified": true,
  "facebookId": "0",
  "failedLoginAttempts": 0,
  "firstName": "Test",
  ...
}
```

BATCHING REQUESTS

- Another format for invoking the Service method is by passing an object with the keys as the service to call, and the value as the service configuration object.

```
Liferay.Service(
  {
    '/user/get-user-by-email-address': {
      companyId: Liferay.ThemeDisplay.getCompanyId(),
      emailAddress: 'test@liferay.com'
    }
  },
  function(obj) {
    console.log(obj);
  }
);
```

INVOKING MULTIPLE SERVICES

- With this format, you can actually invoke multiple services with the same request by passing in an array of service objects.

```
Liferay.Service(
  ['/user/get-user-by-email-address': {
    companyId: Liferay.ThemeDisplay.getCompanyId(),
    emailAddress: 'test@liferay.com'
  },
  {
    '/role/get-user-roles': {
      userId: Liferay.ThemeDisplay.getUserId()
    }
  }
],
function(obj) {
  // obj is now an array of response obejcts
  // obj[0] == /user/get-user-by-email-address data
  // obj[1] == /role/get-user-roles data
  console.log(obj);
}
);
```

NESTING REQUESTS

- ❖ With nested service calls, you can bind information from related objects together in a JSON object.
- ❖ You can call other services within the same HTTP request and nest returned objects in a convenient way.
- ❖ You can use variables to reference objects returned from service calls. Variable names must start with a dollar sign, \$.
- ❖ In this example, we will retrieve user data with /user/get-user-by-id, and use the contactId returned from that service to then invoke /contact/get-contact in the same request.

NESTING REQUESTS EXAMPLE

- ❖ You must flag parameters that take values from existing variables. To flag a parameter, insert the @ prefix before the parameter name.

```
Liferay.Service(
  {
    "$user = /user/get-user-by-id": {
      "userId": Liferay.ThemeDisplay.getUserId(),
      "$contact = /contact/get-contact": {
        "@contactId": "$user.contactId"
      }
    },
    function(obj) {
      console.log(obj);
    }
  );
);
```

NESTING REQUESTS RESULTS

- Here is what the response data would look like for that request.

```
{  
    "agreedToTermsOfUse": true,  
    "comments": "",  
    "companyId": "20116",  
    "contactId": "20157",  
    "createDate": 1471990639779,  
    "defaultUser": false,  
    "emailAddress": "test@liferay.com",  
    "emailAddressVerified": true,  
    ...  
    "contact": {  
        "accountId": "20118",  
        "birthday": 0,  
        "classNameId": "20087",  
        "classPK": "20156",  
        "companyId": "20116",  
        ...  
    }  
}
```

FILTERING RESULTS

- If you don't want all the properties returned by a service, you can define a whitelist of properties.
- Only the specific properties you request in the object are returned from your web service call.
- Here's how you whitelist the properties you need:

```
Liferay.Service(  
    {  
        '$user[emailAddress,firstName] = /user/get-user-by-id': {  
            userId: Liferay.ThemeDisplay.getUserId()  
        }  
    },  
    function(obj) {  
        console.log(obj);  
    }  
);
```

SPECIFYING WHITELIST PROPERTIES

- ❖ To specify whitelist properties, you simply place the properties in square brackets (e.g., [whiteList]) immediately following the name of your variable.
- ❖ Here is what the filtered response looks like.

```
{  
    "firstName": "Test",  
    "emailAddress": "test@liferay.com"  
}
```

INNER PARAMETERS

- ❖ When you pass in an object parameter, you'll often need to populate its inner parameters (i.e., fields).
- ❖ Consider a default parameter `serviceContext` of type `ServiceContext`.
- ❖ To make an appropriate call to JSON web services, you might need to set `serviceContext` fields such as `scopeGroupId`.

```
Liferay.Service(  
    '/example/some-web-service',  
    {  
        serviceContext: {  
            scopeGroupId: 123  
        }  
    },  
    function(obj) {  
        console.log(obj);  
    }  
);
```

Notes:



LIFERAY INTERNAL APIs

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

LIFERAY APIs

- ❖ The Liferay JavaScript object is populated with some helpful tools you're able to use.
- ❖ Below, you'll find some of the various APIs on the object.

LIFERAY.THEMEDISPLAY

```
Liferay.ThemeDisplay = {
    <c:if test="<% layout != null %>">
        getLayoutId: function() {
            return '<%= layout.getLayoutId() %>';
        },
        getLayoutRelativeURL: function() {
            return '<%= PortalUtil.getLayoutRelativeURL(layout, themeDisplay) %>';
        },
        getLayoutURL: function() {
            return '<%= PortalUtil.getLayoutURL(layout, themeDisplay) %>';
        },
        getParentLayoutId: function() {
            return '<%= layout.getParentLayoutId() %>';
        },
        ...
    </c:if>
    ...
};
```

LIFERAY.AUI

```
Liferay.AUI = {
    Object {}
    getAvailableLangPath: function() {...}
    getCombine: function() {...}
    getComboPath: function() {...}
    getDateFormat: function() {...}
    getEditorCKEditorPath: function() {...}
    getFilter: function() {...}
    getFilterConfig: function() {...}
    getJavaScriptRootPath: function() {...}
    getLangPath: function() {...}
    getPortletRootPath: function() {...}
    getStaticResourceURLParams: function() {...}
}
```

LIFERAY.BROWSER

```
var Liferay = Liferay || {};
Liferay.Browser = {
    acceptsGzip: function() {
        return <%= BrowserSnifferUtil.acceptsGzip(request) %>;
    },
    getMajorVersion: function() {
        return <%= BrowserSnifferUtil.getMajorVersion(request) %>;
    },
    getRevision: function() {
        return '<%= BrowserSnifferUtil.getRevision(request) %>';
    },
    getVersion: function() {
        return '<%= BrowserSnifferUtil.getVersion(request) %>';
    },
    isAir: function() {
        return <%= BrowserSnifferUtil.isAir(request) %>;
    },
    ...
};
```

WWW.LIFERAY.COM



LIFERAY.PROPSVALUES

```
Liferay.PropsValues = {
    JAVASCRIPT_SINGLE_PAGE_APPLICATION_TIMEOUT: <%= PrefsPropsUtil.getInteger(
        themeDisplay.getCompanyId(), PropsKeys.
        JAVASCRIPT_SINGLE_PAGE_APPLICATION_TIMEOUT, PropsValues.
        JAVASCRIPT_SINGLE_PAGE_APPLICATION_TIMEOUT) %>,
    NTLM_AUTH_ENABLED: <%= PrefsPropsUtil.getBoolean(themeDisplay.getCompanyId(),
        PropsKeys.NTLM_AUTH_ENABLED, PropsValues.NTLM_AUTH_ENABLED) %>,
    UPLOAD_SERVLET_REQUEST_IMPL_MAX_SIZE: <%= PrefsPropsUtil.getLong(PropsKeys.
        UPLOAD_SERVLET_REQUEST_IMPL_MAX_SIZE, PropsValues.
        UPLOAD_SERVLET_REQUEST_IMPL_MAX_SIZE) %>
};
```

WWW.LIFERAY.COM



LIFERAY.PORTLETKEYS

```
Liferay.PortletKeys = {
    DOCUMENT_LIBRARY: '<%= PortletKeys.DOCUMENT_LIBRARY %>',
    DYNAMIC_DATA_MAPPING:
    'com_liferay_dynamic_data_mapping_web_portlet_DDMPortlet',
    ITEM_SELECTOR: '<%= PortletKeys.ITEM_SELECTOR %>'
};
```

LIFERAY.DATA

```
Liferay.Data = Liferay.Data || {};
Liferay.Data.NAV_SELECTOR = '#navigation';
Liferay.Data.NAV_SELECTOR_MOBILE = '#navigationCollapse';
Liferay.Data.isCustomizationView = function() {
    return <%= (layoutTypePortlet.isCustomizable() && LayoutPermissionUtil.contains(permissionChecker, layout, ActionKeys.CUSTOMIZE)) %>;
};
Liferay.Data.notices = [
    null
    <c:if test="<%= permissionChecker.isOmniadmin() && PortalUtil.isUpdateAvailable() %>">
        ,
        {
            content: '<a class="update-available" href="<%= themeDisplay.getURLUpdateManager() %>"><liferay-ui:message key="updates-are-available-for-liferay" /></a>',
            toggleText: false
        }
    </c:if>
    ...
];
```

LIFERAY.AUTHTOKEN

```
Liferay.authToken = '<%= AuthTokenUtil.getToken(request) %>';
Liferay.currentURL = '<%= HtmlUtil.escapeJS(currentURL) %>';
Liferay.currentURLEncoded = '<%= HtmlUtil.escapeJS(HttpUtil.encodeURL(currentURL))
%>';
```

LIFERAY.CURRENTURL

```
Liferay.currentURL = '<%= HtmlUtil.escapeJS(currentURL) %>';
Liferay.currentURLEncoded = '<%= HtmlUtil.escapeJS(HttpUtil.encodeURL(currentURL))
%>';
```

LIFERAY.CURRENTURLENCODED

```
Liferay.currentURLEncoded = '<%= HtmlUtil.escapeJS(HttpUtil.encodeURL(currentURL))%>';
```

LIFERAY.PORTLET

- ❖ Custom applications that implement their own module will add to the pre-existing Liferay.Portlet object.
- ❖ An example of this can be seen in the Google Maps Portlet here:
<https://github.com/liferay/liferay-portal/blob/master/modules/apps/google-maps/google-maps-web/src/main/resources/META-INF/resources/js/main.js>
- ❖ Adding the module Liferay.Portlet makes it available globally.

LIFERAY.PORTLETURL

- ❖ This provides a way to create Liferay PortletURLs such as `actionURL`, `renderURL`, and `resourceURL` through JavaScript.
- ❖ Here is an example:

```
var portletURL = Liferay.PortletURL.createURL(themeDisplay.  
getURLControlPanel());  
  
portletURL.setDoAsGroupId('true');  
portletURL.setLifecycle(Liferay.PortletURL.ACTION_PHASE);  
portletURL.setParameter('cmd', 'add_temp');  
portletURL.setParameter('javax.portlet.action',  
'/document_library/upload_file_entry');  
portletURL.setParameter('p_auth', Liferay.authToken);  
portletURL.setPortletId(Liferay.PortletKeys.DOCUMENT_LIBRARY);
```

LIFERAY.MAPS

- ❖ In DXP, the new Loader makes it possible to add some third-party modules and dependencies for your modules.
- ❖ These two objects make it possible to see what modules are being loaded into the application for each module.
- ❖ Here is an example of Liferay.MAPS:

```
Liferay.MAPS = {  
    classic-theme: "classic-theme@1.0.2"  
    com.liferay.announcements.web: "com.liferay.announcements.web@1.0.5"  
    com.liferay.application.list.taglib: "com.liferay.application.list.  
taglib@2.0.4"  
    com.liferay.asset.browser.web: "com.liferay.asset.browser.web@1.0.6"  
    com.liferay.asset.categories.admin.web: "com.liferay.asset.categories.  
admin.web@1.0.9"  
    com.liferay.asset.categories.navigation.web: "com.liferay.asset.categories.  
navigation.web@1.0.4"  
    com.liferay.asset.display.web: "com.liferay.asset.display.web@1.0.1"  
    ...  
}
```

LIFERAY MODULES

- ❖ This shows all of the included module names, the module they belong to, and that module's version number.
- ❖ Here is an example of Liferay.MODULES:

```
Liferay.MODULES = {  
    ...  
    frontend-image-editor-capability-crop@1.0.2/CropHandles.es: {...}  
    frontend-image-editor-capability-crop@1.0.2/CropHandles.soy: {  
        conditionalMark: false,  
        dependencies: ["exports", "metal-component/src/Component", "metal-  
            soy/src/Soy"]  
    }  
    "frontend-image-editor-capability-crop@1.0.2/CropHandles.soy": {...}  
    ...  
}
```

- ❖ This object displays each module and its dependencies.



JAVASCRIPT: SINGLE PAGE APPLICATIONS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS A SINGLE PAGE APPLICATION?

- A *Single Page Application* (SPA) is a web application that loads a single HTML page and dynamically updates that page as the user interacts and navigates through the site.
 - Using AJAX, we can update only the portions of the page that change, instead of having to reload the entire page.
- There are many benefits to using an SPA, including:
 - A more native app-like experience for the user
 - Decreased load time and amount of data that needs to be transferred from server to client
 - Improved user experience because the data can be loaded in the background

WHAT IS SENNA.JS?

- ❖ *Senna.js* makes the SPA magic happen in Liferay.
- ❖ It is a lightweight, fast, open-source *Single Page Application* engine.
- ❖ It was built by a team of developers here at Liferay as well as a bunch of great contributors.
- ❖ *Senna.js* has been integrated into DXP to leverage the benefits of an SPA.
- ❖ For detailed documentation and examples, you can check out <http://sennajs.com/docs>.

HOW DOES IT WORK?

- ❖ After the SPA is initially loaded, all subsequent navigations are handled without a full page reload.
- ❖ Additional content is loaded using XMLHttpRequest via History API, which is able to update the URL without refreshing the page, therefore making it so your dynamic site can be shared and bookmarked.
- ❖ Having a URL for each state of your page also makes it so that the content can be fully crawled by search engines.
- ❖ There are many features *Senna.js* provides to make sure our SPAs meet these needs.

SENNA.JS FEATURES

- **SEO & Bookmarkability:** Sharing or bookmarking a link should always display the same content. Search engines are able to index that same content.
- **Hybrid rendering:** Ajax + server-side rendering allows disabling pushState at any time, allowing progressive enhancement.
 - The way you render the server-side doesn't matter. It can be simple HTML fragments or even template views.
- **State retention:** Scrolling, reloading, or navigating through the history of the page should get back to where it was.
- **UI feedback:** When some content is requested, it indicates to the user that something is happening.
- **Pending navigations:** Blocks UI rendering until data is loaded, then displays the content at once
 - It's important to give some UX feedback during loading.

ADDITIONAL SENNA.JS FEATURES

- **Timeout detection:** Timeout if the request takes too long to load or when navigating to a different link while another request is pending
- **History navigation:** By using History API, you can manipulate the browser history, so you can use the browser's back and forward buttons.
- **Cacheable screens:** Once you load a certain surface, this content is cached in memory and is retrieved later on without any additional request.
- **Page resources management:** Evaluate scripts and stylesheets from dynamically loaded resources.
 - Additional content loaded using XMLHttpRequest can be appended to the DOM. For security reasons, some browsers won't evaluate `<script>` tags from the new Fragment.
 - The SPA engine should handle extracting scripts from the content and parsing them, respecting the browser contract for loading scripts.

ENABLING SPA IN LIFERAY

- Enabling SPA in Liferay is simple.
- All that is needed is to have the `frontend-js-spa-web` module deployed and enabled.
 - This module is included with Liferay by default.
- SPA is also enabled by default and requires no changes to your workflow or existing code.

CUSTOMIZING SPA SETTINGS

- It's easy to customize the behavior of your *Single Page Application*.
- Depending on what you need to configure, you can customize SPA options in one of two places:
 - Caching and timeout settings can be configured in *System Settings*.
 - Options for disabling SPAs are set within specific elements.
- To configure settings via *System Settings* in Liferay, go to *Control Panel*→*Configuration*→*System Settings*, click the *Foundation* tab, and then click on *Frontend SPA Infrastructure*.



This configuration was not used yet. The values shown are the default.

All fields marked with * are required.

Cache Expiration Time *
Determines how long to cache the SPA's static assets after they were last modified. The longer it is, the better the performance.

-1

Request Timeout Time *
Determines how long to wait for a response from the SPA. If the response is not received within this time, the request is canceled.

User property:com.liferay.portal.json.rpc.single.page.application.timeout

User Notification Timeout *
Determines how long to wait for a user notification to be acknowledged before canceling the request. The user can cancel the notification at any time.

30000

Save **Cancel**

SPA SETTINGS OPTIONS

- You will find a number of options in the *Frontend SPA Infrastructure* section of *System Settings*:
 1. **Cache Expiration Time:** The time, in minutes, in which the SPA cache is cleared.
 - A zero value means the cache should never expire during SPA navigation.
 - A negative value means the cache should be disabled.
 2. **Request Timeout Time:** The time, in milliseconds, in which a SPA request times out.
 - A zero value means the request should never timeout.
 3. **User Notification Time:** The time, in milliseconds, in which a notification is shown to the user stating that the request is taking longer than expected.
 - A zero value means no notification should be shown.

DISABLING SPA IN LIFERAY

- To disable SPA across all of Liferay, you can add the following line to the `portal-ext.properties` file:
`javascript.single.page.application.enabled = false`

DISABLING SPA IN SPECIFIC ELEMENTS

- ❖ If there is a certain application or page that you don't want to be part of the SPA, you have some options:
 - Blacklist the application to disable SPA for the entire application
 - Use the `data-senna-off` annotation to disable SPA for a specific form or link
- ❖ To blacklist an application from SPA, open your application class and set the `_singlePageApplication` property to false:

```
_singlePageApplication = false;
```

 - If you prefer, you can set this property to false in your `portlet.xml` instead by adding the following property to the `<portlet>` section:

```
<single-page-application>false</single-page-application>
```
- ❖ To disable SPA for a specific form or link, add the `data-senna-off` attribute to the element and set the value to `true`. For example:

```
<a data-senna-off="true" href="/pages/page2.html">Page 2</a>
```

LEVERAGING THE SPA LIFECYCLE

- ❖ Sometimes it is necessary to know the navigation status.
- ❖ *Senna.js* makes this easy by exposing lifecycle events that represent state changes in the application. The available events are:
 - `beforeNavigate`: Fires before navigation starts. Event payload: { `path: '/pages/page1.html'`, `replaceHistory: false` }
 - `startNavigate`: Fires when navigation begins. Event payload: { `form: '<form name="form"></form>'`, `path: '/pages/page1.html'`, `replaceHistory: false` }
 - `endNavigate`: Fired after the content has been retrieved and inserted onto the page. Event payload: { `form: '<form name="form"></form>'`, `path: '/pages/page1.html'` }
- ❖ These events can be leveraged easily by listening for them on the Liferay global object.

```
Liferay.on('beforeNavigate', function(event) {
    alert("Get ready to navigate to " + event.path);
});
```

GLOBAL LISTENERS

- ❖ SPAs provide several improvements that benefit your site, but there may be some additional maintenance required as a result.
- ❖ In a traditional navigation scenario, every page refresh resets everything, so you don't have to worry about what's left behind.
- ❖ In a SPA scenario however, global listeners such as `Liferay.on` or `Liferay.after` or `body` delegates can become problematic.
- ❖ Every time you execute these global listeners, you add yet another listener to the globally persisted Liferay object which results in multiple invocations of those listeners.

DETACHING EVENT LISTENERS

- ❖ To prevent this, you need to listen to the navigation event in order to detach your listeners.
- ❖ For example, you would use the following code to detach the event listeners of a global category event:

```
var onCategory = function(event) {...};

var clearPortletHandlers = function(event) {
    if (event.portletId === '<%= portletDisplay.getRootPortletId() %>') {
        Liferay.detach('onCategoryHandler', onCategory);
        Liferay.detach('destroyPortlet', clearPortletHandlers);
    }
};

Liferay.on('category', onCategory);
Liferay.on('destroyPortlet', clearPortletHandlers);
```

Notes:



LOADING SCRIPTS WITH THE AUI SCRIPT TAG

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS THE AUI SCRIPT TAG?

- The AUI script tag is a taglib for JSPs that allows you to load JavaScript in script tags on the page and ensure certain resources are loaded before executing.

AUI SCRIPT OPTIONS

- ❖ Here is a list of attribute options:
 1. **require**: Requires an AMD module that will be loaded with the Liferay AMD Module Loader
 2. **use**: Uses an AlloyUI/YUI module that will be loaded via the YUI loader
 3. **position**: The position the script tag is put in the page; can be `inline` or `auto`
 4. **sandbox**: If set to true, it will wrap the script tag in an anonymous function. Also, `$` and `_` will be defined for convenient use of jQuery and underscore.

USING THE USE ATTRIBUTE

- ❖ Here is an example of the `use` attribute:

```
<aui:script use="aui-base">
    A.one('#someNodeId').on(
        'click',
        function(event) {
            alert('Thank you for clicking.')
        }
    );
</aui:script>
```

USING THE REQUIRE ATTRIBUTE

- Here is an example of the *require* attribute:

```
<aui:script require="metal-clipboard/src/Clipboard">
    new metalClipboardSrcClipboard.default();
</aui:script>
```

Notes:



THEME DISPLAY OBJECT

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

LIFERAY THEME DISPLAY OBJECT IN JAVASCRIPT

- ❖ The `ThemeDisplay` object in Java provides general configuration methods for the platform, providing access to the pages, sites, themes, locales, URLs, and more.
- ❖ This class is an information context object that holds data commonly referred to for various kinds of front-end information.
- ❖ There is also a global variable in JavaScript, `Liferay.ThemeDisplay` that contains the same information.
- ❖ The object contains the following methods.

THEME DISPLAY METHODS (I)

```
getLayoutId: function() {...},  
getLayoutRelativeURL: function() {...},  
getLayoutURL: function() {...},  
getParentLayoutId: function() {...},  
isControlPanel: function() {...},  
isPrivateLayout: function() {...},  
isVirtualLayout: function() {...},  
getBCP47LanguageId: function() {...},  
getCDNBaseUrl: function() {...},  
getCDNDynamicResourcesHost: function() {...},  
getCDNHost: function() {...},  
getCompanyGroupId: function() {...},  
getCompanyId: function() {...},  
getDefaultValueId: function() {...},  
getDoAsUserIdEncoded: function() {...},  
getLanguageId: function() {...},  
getParentGroupId: function() {...},  
getPathContext: function() {...},  
getPathImage: function() {...},  
getPathJavaScript: function() {...},
```

THEME DISPLAY METHODS (II)

```
getPathMain: function() {...},  
getPathThemeImages: function() {...},  
getPathThemeRoot: function() {...},  
getPlid: function() {...},  
getPortalURL: function() {...},  
getScopeGroupId: function() {...},  
getScopeGroupIdOrLiveGroupId: function() {...},  
getSessionId: function() {...},  
getSiteGroupId: function() {...},  
getURLControlPanel: function() {...},  
getURLHome: function() {...},  
getUserID: function() {...},  
getUserName: function() {...},  
isAddSessionIdToURL: function() {...},  
isFreeformLayout: function() {...},  
isImpersonated: function() {...},  
isSignedIn: function() {...},  
isStateExclusive: function() {...},  
isStateMaximized: function() {...},  
isStatePopUp: function() {...}
```

THEME DISPLAY EXAMPLE

- ❖ Here is an example of how it can be used:

```
var themeImageHref = Liferay.ThemeDisplay.getPathThemeImages() +  
'/file_system/large/default.png';
```
- ❖ You can find all of the default variables in the `build/init.ftl` file.
- ❖ All custom variables can be created in the `init_custom.ftl` as seen in our theme.

Notes:



MODERN WEB EXPERIENCES: ECMASCIPT 2015 FEATURES

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

ECMASCRIPT 2015

- ❖ *ECMAScript 2015*, previously known as ES6, is the latest version of the ECMAScript standard.
- ❖ In DXP, you can now write JavaScript that adheres to the new ECMAScript 2015 syntax, leverage ES2015 advanced features in your modules, and publish them.
- ❖ ECMAScript 2015 is a significant update to the language, and the first update to the language since ES5 was standardized in 2009.

```
export function sum(x, y) {  
    return x + y;  
}  
export var pi = 3.141593;
```

WHAT'S NEW IN ECMASCIPT 2015?

- ❖ Here is a short list of some of the benefits of using ECMAScript 2015:
 - Class syntax like other OO languages
 - Classes support prototype-based inheritance, super calls, instance and static methods, and constructors.
 - Arrow method syntax
 - var odds = numbers.map(v => v + 1);
 - Modules
 - Let and const declarations
 - Language-level support for modules for component definition
 - Codifies patterns from popular JavaScript module loaders and specifications like asynchronous module defintion (AMD).
- ❖ Let's take a look at examples of each of these features.

CLASSES

- ❖ Classes with constructors and inheritance:

```
class Car {  
    constructor(make) { //constructors!  
        this.currentSpeed = 25;  
    }  
  
    printCurrentSpeed(){  
        console.log('current speed: ' + this.currentSpeed + ' mph.');//  
    }  
}  
class RaceCar extends Car { //inheritance  
    constructor(make, topSpeed) {  
        super(make);  
        this.topSpeed = topSpeed;  
    }  
  
    goFast(){  
        this.currentSpeed = this.topSpeed;  
    }  
}
```

ARROW FUNCTIONS

- ❖ **Arrow Functions**, which make anonymous functions easier:

```
setTimeout(() => {  
    alert("Hello from an arrow function!")  
}, 1000);
```

MODULES

- ❖ **Modules** give you the ability to create, load, and manage dependencies via the new import and export keywords:

```
import $ from 'lib/jquery';
```

- ❖ To make modules discoverable, you need to write a package.json file with the name and version of your ECMAScript 2015 module.

LET AND CONST DECLARATIONS

- ❖ **let** and **const** declarations:

```
//let
function letTest() {
    let x = 1; // let declares a frame scope local variable
    if (true) {
        let x = 2; // different variable
        console.log(x); // 2
    }
    console.log(x); // 1
}

//const
const ALWAYS_SEVEN = 7;

// this will throw an error
ALWAYS_SEVEN = 8;
```

ECMASCRIPT 2015 BROWSER SUPPORT

- ❖ Most modern browsers support ECMAScript 2015. Liferay DXP comes with the ability to transpile ECMAScript 2015 code.
- ❖ To use ECMAScript 2015 syntax and advanced features, you need to make the following adjustments to your JavaScript files:
 1. Files containing ECMAScript 2015 code that needs to be transpiled should end in `.es.js`.
 2. Import the `polyfillBabel` class from the `polyfill-babel` module to use advanced features like generators.

```
import polyfillBabel from 'polyfill-babel'
```
- ❖ With Themes, you can also take advantage of *Liferay Theme ES2015 Hook*: <https://www.npmjs.com/package/liferay-theme-es2015-hook>
- ❖ *ECMA-262 6th Edition, The ECMAScript 2015 Language Specification* can be found here: <http://www.ecma-international.org/ecma-262/6.0/>

Notes:



BUILDING COMPONENTS: METAL.JS

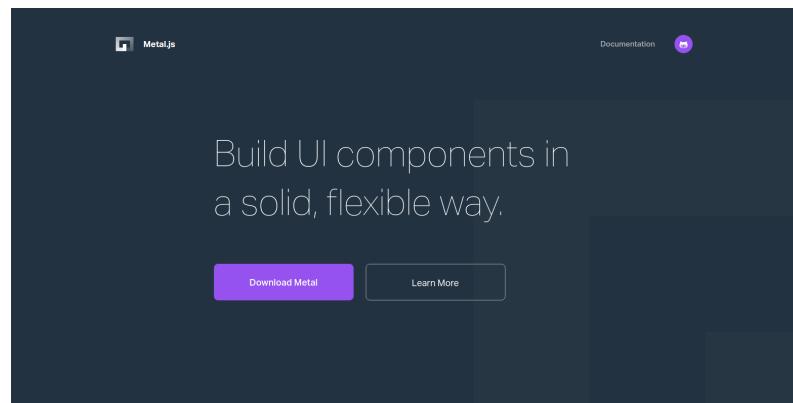
Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

BUILD UI COMPONENTS IN A SOLID, FLEXIBLE WAY

- Metal.js is a lightweight, easy-to-use JavaScript framework that integrates with templating languages to help you create UI Components.

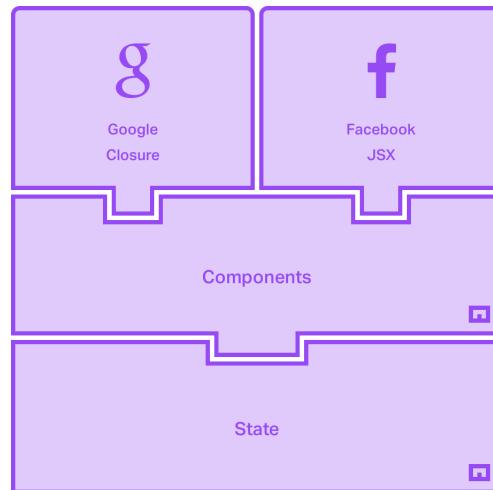


ARCHITECTURE

- ❖ Metal.js's main classes are *State* and *Component*.
- ❖ *Component* actually extends from *State*, so it contains all its features.
- ❖ The main difference between the two is that *Component*'s extra features are related to rendering.
- ❖ If your module doesn't do any rendering, you could just use *State* directly.
- ❖ *Component* will work better for you if your module needs rendering logic.
- ❖ Many people have their favorite way of dealing with rendering logic.
- ❖ Some prefer to use template languages that completely separate the rendering logic from the business logic, while others like to keep both close together in the same file.
- ❖ Metal.js doesn't force developers to go with only one option.

STRUCTURE VISUALIZED

- ❖ Here's a visualization of the structure:



TEMPLATES

- ❖ Metal.js offers integration points with both Closure Templates from Google (Soy Templates) and JSX from Facebook.
- ❖ It's possible to add more options, as the Rendering Layer is customizable.
- ❖ A Soy Template in Metal.js may look like this:

```
{template .render}
  // ...
  <button onClick="${close}" type="button" class="close">
    // ...
{/template}
```

AN ECMASCIPT 2015 COMPONENT IN METAL.JS

- ❖ A Metal.js component written in *ECMAScript 2015* may look like this:

```
class MyComponent extends Component {
  created() {
    //do some things
  }

  disposed() {
    //do some other things
  }
}
```

- ❖ These just scratch the surface of the new features in *ECMAScript 2015* that can be leveraged in Metal.js.

Notes:



ALLOYUI

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

WHAT IS ALLOYUI?

- AlloyUI is a JavaScript framework, built on top of YUI, that provides the following:
 - A large set of components, from various input controls to complex UIs like tree views and data tables
 - A uniform API for using and creating JavaScript modules
 - Modular loading of JavaScript components for scalable applications
 - Bootstrap 3 for a consistent UI look

ALLOYUI USAGE IN LIFERAY

- ❖ AlloyUI has been included by default with Liferay since version 6.0.
- ❖ Taglibs were built around AlloyUI components to make integration simpler for front-end and back-end developers.
- ❖ The majority of Liferay's JavaScript is patterned after AlloyUI's module pattern.
- ❖ Liferay is moving in a direction that means developers have full control over the front-end libraries that are loaded onto the page, making the individual components more reusable and modular.
- ❖ So where does AlloyUI stand in 7.0?

ALLOYUI STATUS BEGINNING WITH 7.0

- ❖ AlloyUI is being sunsetted and deprecated, making it no longer recommended for new development.
- ❖ For those already using AlloyUI, there will be continued support, but AlloyUI is in maintenance mode.
- ❖ AlloyUI is included by default, so existing based code will still continue to work.
 - If you wrote AlloyUI modules, or extended existing AlloyUI modules, there is no need for rewriting, as your code will still work.
- ❖ This framework will continue to be available in future versions of Liferay to give developers time to migrate to their front-end technologies of choice.

Notes:

Appendix - Portlets



APPLICATIONS: INTRODUCTION

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

TYPES OF APPLICATIONS IN LIFERAY DXP

- In Liferay, it is possible to create:
 - Liferay MVC Application
 - Traditional Model View Controller-style application that helps separate concerns of View, Render, Action, and Resource requests into different Java classes for more modularity.
 - Soy Portlet
 - You can look at the documentation on Soy Portlets for more information.
 - Spring MVC
 - Spring MVC is the web component of Spring's framework.
 - It provides a rich functionality for building robust Web Applications.
 - The Spring MVC Framework is designed so that every piece of logic and functionality is highly configurable.
 - In DXP, you can still use features like the Service Builder.
 - JavaServer Faces
 - Liferay Faces is an open-source umbrella project that provides support for the JavaServer™ Faces (JSF) standard in webapp and portlet projects.

CREATING AN APPLICATION USING BLADE CLI

- ❖ Blade CLI is a command line Java tool that can be used to help Bootstrap Liferay module development.
- ❖ It is installed using a Java package manager called jpm4j.
- ❖ Blade CLI makes it easy to create modules in Liferay.
- ❖ An MVC Application can be created with the following command:
`blade create -t mvcportlet -p [package name] -c [class name] [project name]`
- ❖ Here is a working example:
`blade create -t mvcportlet -p com.liferay.docs.mvcportlet -c MyMvcPortlet my-mvc-portlet-project`

FOLDER STRUCTURE

- ❖ From here, you can use the Liferay MVC Framework for creating your application.

```
- my-mvc-portlet-project
  - src
    - main
      - java
        - com/liferay/docs/mvcportlet
          - MyMvcPortlet.java
    - resources
      - content
        - Language.properties
      - META-INF
        - resources
          - css
          - js
          - init.jsp
          - view.jsp
    - bnd.bnd
    - build.gradle
```

START BUILDING

- ❖ Having the following code in the MyMVCPortlet.java is the starting point for building your application:

```
@Component(  
    immediate = true,  
    property = {  
        "com.liferay.portlet.css-class-wrapper=portlet-my-mvc-portlet",  
        "com.liferay.portlet.display-category=category.sample",  
        "com.liferay.portlet.footer-portlet-javascript=/js/main.js",  
        "com.liferay.portlet.header-portlet-css=/css/main.css",  
        "com.liferay.portlet.preferences-owned-by-group=true",  
        "com.liferay.portlet.private-request-attributes=false",  
        ...  
    },  
    service = Portlet.class  
)  
public class MyMvcPortlet extends MVCPortlet {  
    ...  
}
```

Notes:



APPLICATION DISPLAY TEMPLATE USAGE IN CUSTOM APPLICATIONS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

USING AN APPLICATION DISPLAY TEMPLATE IN A CUSTOM APPLICATION

- To leverage the *Application Display Template (ADT)* API, you need to follow several steps:
- These steps involve the following:
 - Registering your application to use ADTs
 - Defining permissions
 - Exposing the ADT functionality to users
- You'll walk through these steps next.

REGISTERING THE PORTLETDISPLAYTEMPLATEHANDLER

➤ Let's take a look at these steps.

1. Create and register a custom `*PortletDisplayTemplateHandler` component. Liferay provides the `BasePortletDisplayTemplateHandler` as a base implementation for you to extend.
 - <https://docs.liferay.com/portal/7.0/javadocs/portal-kernel/com/liferay/portal/kernel/portletdisplaytemplate/BasePortletDisplayTemplateHandler.html>
2. You can check the `TemplateHandler` interface Javadoc to learn about each template handler method. As an example of the `*PortletDisplayTemplateHandler` implementation, you can look at the `WikiPortletDisplayTemplateHandler.java`:
 - <https://github.com/liferay/liferay-portal/blob/master/modules/apps/collaboration/wiki/wiki-web/src/main/java/com/liferay/wiki/web/internal/portlet/template/WikiPortletDisplayTemplateHandler.java>

CONFIGURING PERMISSIONS

1. Since the ability to add ADTs is new to your application, you must configure permissions so that administrative users can grant permissions to the roles that will be allowed to create and manage ADTs.
2. Add the action key `ADD_PORTLET_DISPLAY_TEMPLATE` to your application's `/src/main/resources/resource-actions/default.xml` file.

ADDING ADD_PORTLET_DISPLAY_TEMPLATE

1. As an example:

```
<!DOCTYPE resource-action-mapping PUBLIC "-//Liferay//DTD Resource Action
Mapping 7.0.0//EN" "http://www.liferay.com/dtd/liferay-resource-action
-mapping_7_0_0.dtd">
<resource-action-mapping>
    ...
    <portlet-resource>
        <portlet-name>yourportlet</portlet-name>
        <permissions>
            <supports>
                <action-key>ADD_PORTLET_DISPLAY_TEMPLATE</action-key>
                <action-key>ADD_TO_PAGE</action-key>
                <action-key>CONFIGURATION</action-key>
                <action-key>VIEW</action-key>
            </supports>
            ...
        </permissions>
    </portlet-resource>
    ...
</resource-action-mapping>
```

EXPOSING ADT OPTIONS TO USERS

- ▶ Next, you can expose the ADT option to your users.
- ▶ Just include the `<liferay-ui:ddm-template-selector>` tag in the JSP file you're using to control your application's configuration.
- ▶ As an example JSP, see the Wiki application's `configuration.jsp`:
 - ▶ <https://github.com/liferay/liferay-portal/blob/master/modules/apps/collaboration/wiki/wiki-web/src/main/resources/META-INF/resources/wiki/configuration.jsp>

ADDING ADTS TO A APPLICATION'S VIEW LAYER

- ❖ Next, you can extend your View code to render your application with the selected ADT.
- ❖ This allows you to decide which part of your View will be rendered by the ADT and what will be available in the template context.
- ❖ You'll need to set up what you want to allow the ADT to mark up and pass that to <liferay-ddm:template-renderer> as entries.
- ❖ As an example JSP, see the Wiki application's view.jsp:
 - <https://github.com/liferay/liferay-portal/blob/master/modules/apps/collaboration/wiki/wiki-web/src/main/resources/META-INF/resources/wiki/view.jsp>
- ❖ Then you can create your own scripts to change the display of your application.
- ❖ You can experiment by adding your own custom ADT.

Notes:



LOCALIZATION

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

LOCALIZATION IN DXP

- ❖ Liferay makes it easy to support translation of your application into any language.
- ❖ The process involves creating language keys that correspond to specific translations.
- ❖ These key/value pairs belong in language properties files.

WHAT ARE LANGUAGE PROPERTIES FILES?

- ❖ Language properties files are documents containing your language keys and translations.
- ❖ First, it is necessary to create a default language properties file named `Language.properties`.
- ❖ For each language you'd like to support, you will need an additional file that is named `Language_xx.properties`, where `xx` is the language abbreviation.
- ❖ For example, if you'd like to support English, French, and Spanish in your application, you would have files named:

`Language.properties`
`Language_fr.properties`
`Language_es.properties`

DEFAULT LOCALES

- ❖ Some locales are available by default in Liferay.
- ❖ Look in the `portal.properties` file to find them.

```
locales=ar_SA,eu_ES,bg_BG,ca_AD,ca_ES,zh_CN,zh_TW,hr_HR,cs_CZ,da_DK,nl_NL,  
nl_BE,en_US,en_GB,en_AU,et_EE,fi_FI,fr_FR,fr_CA,gl_ES,de_DE,el_GR,  
iw_IL,hi_IN,hu_HU,in_ID,it_IT,ja_JP,ko_KR,lo_LA,lt_LT,nb_NO,fa_IR,  
pl_PL,pt_BR,pt_PT,ro_RO,ru_RU,sr_RS,sr_RS_latin,sl_SI,sk_SK,es_ES,  
sv_SE,tr_TR,uk_UA,vi_VN
```

LANGUAGE FILES

- ❖ In an application with only one module that holds all the views (for example, all its JSPs) and application components, just create an `src/main/resources/content` folder in that module and place all your `Language_xx.properties` there.
- ❖ After that, make sure any application components (the `@Component` annotation in your `-Portlet` classes) in the module include this property:
`"javax.portlet.resource-bundle=content.Language"`
- ❖ Providing translated language properties files and specifying the `javax.portlet.resource-bundle` property in your application component is all you need to do to have your language keys translated.
- ❖ Then, when the locale is changed in Liferay, your application's language keys will be automatically translated.

LANGUAGE FILE FORMAT

- ❖ Language files follow the standard properties file format. They should look something like:

```
hello=Hello  
welcome-to-liferay=Welcome to Liferay  
please-click-here-to-continue=Please click here to continue.
```

AUTOMATICALLY GENERATING LANGUAGE FILES

- ❖ Instead of manually creating a language properties file for each locale that's supported by Liferay, you can get them all automatically generated for you with one command.
- ❖ The same command also propagates the keys from the default language file to all of the translation files.

MICROSOFT'S TRANSLATOR API

- ❖ You can take a few additional steps and get automatic translations using Microsoft's Translator API.
 1. Make sure your module's build includes the `com.liferay.lang.builder` plugin by putting the plugin in build script classpath.
 2. Make sure you have a default `Language.properties` file in `src/main/content`.
 3. Run the gradle `buildLang` task from your project's root directory to generate default translation files.
- ❖ The generated files will contain automatic copies of all the keys and values in your default `Language.properties` files.
- ❖ That way you don't have to worry about manually copying your language keys into all of the files.
- ❖ Just run the `buildLang` task each time you change the default language file.

HOW TO AUTO-GENERATE LANGUAGE FILES

- ❖ If you'd like to use Microsoft's Translator API, you must register your application with Azure DataMarket.

1. Follow the instructions here:

<https://msdn.microsoft.com/en-us/library/hh454950>

2. Make sure the `buildLang` task knows to use your credentials for translation.

3. For security reasons, you probably don't want to pass them directly in your application's build script.

```
buildLang {  
    translateClientId = "my-id"  
    translateClientSecret = "my-secret"  
}
```

- ❖ Then, when you run `buildLang`, translations will be automatically generated.

TRANSLATIONS ON THE FRONT-END

- ❖ There is a Liferay language object defined at `Liferay.Language`.
- ❖ It contains all of the available locales along with a `get` method. When the `get` method is passed, a language key will return the translation in the current locale.

```
Liferay.Language.get('click-here'); //returns "Click Here"
```

Notes:



THEMING APPLICATIONS

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

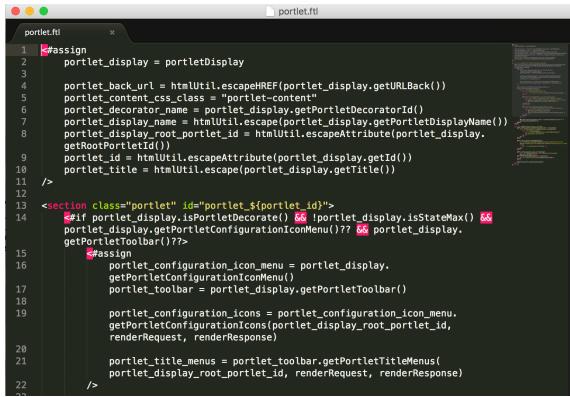
No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

STYLING YOUR APPLICATIONS

- ❖ Liferay themes have the ability to provide additional styles to applications.
- ❖ We've seen how developers can add *application decorators* to provide selectable application wrapper styles.
- ❖ Through the `portlet.ftl` developers can also change markup for the containers that hold applications.

PORLET.FTL

- The default portlet.ftl can be found in the build/templates folder of your theme.
- To modify the HTML markup, developers can copy this into the src/templates folder of their custom theme.



```
portlet.ftl
1  #assign
2   portlet_display = portletDisplay
3
4   portlet_back_url = htmlUtil.escapeREF(portlet_display.getURLBack())
5   portlet_content_css_class = portlet.getTemplateContentCSSClass()
6   portlet_decorator_id = portletDisplay.getPortletDecoratorId()
7   portlet_display_name = htmlUtil.escape(portlet_display.getPortletDisplayName())
8   portlet_display_root_portlet_id = htmlUtil.escapeAttribute(portlet_display,
9     getRootPortletId())
9   portlet_id = htmlUtil.escapeAttribute(portlet_display.getId())
10  portlet_title = htmlUtil.escape(portlet_display.getTitle())
11
12
13 <section class="portlet" id="portlet_${portlet_id}">
14   #if portlet_display.isPortletDecorate() && !portlet_display.isStateMax() &&
14     portlet_display.getPortletConfigurationIconMenu()??
14     portlet_display.getPortletToolbar()?>
15   #assign
16     portlet_configuration_icon_menu = portlet_display.
17     getPortletConfigurationIconMenu()
18     portlet_toolbar = portlet_display.getPortletToolbar()
19
20     portlet_configuration_icons = portlet_configuration_icon_menu.
20     getPortletConfigurationIcons(portlet_display_root_portlet_id,
21     renderRequest, renderResponse)
22
23   portlet_title_menus = portlet_toolbar.getPortletTitleMenus(
23     portlet_display_root_portlet_id, renderRequest, renderResponse)
24
```

UNDERSTANDING THE PORTLET.FTL

- portletDisplay: is fetched from the themeDisplay object and contains information about the application
- portlet_back_url: URL to return to previous page with applicationWindowState is maximized
- portlet_display_name: Custom application name if set
- portlet_id: The id of application (not the same as the application namespace)
- portlet_title: Application name set in Portlet java class (usually from a *Keys.java class)

PORLET.FTL PORTLET HEADER

- ❖ This conditional checks to see the application header should be displayed.
- ❖ It checks to see if there is a application toolbar for the application (Configuration, Permissions, Look and Feel).

```
<#if portlet_display.isPortletDecorate() && !portlet_display.isStateMax() &&  
portlet_display.getPortletConfigurationIconMenu()?? && portlet_display.  
getPortletToolbar()??>
```

PORLET.FTL PORTLET TITLE

- ❖ The application title menu is used in applications that allow for adding resources (Web Content Display, Media Gallery, Documents and Media).
- ❖ It's used to build a menu of items for adding resources.

```
portlet_title_menus = portlet_toolbar.getPortletTitleMenus(  
portlet_display_root_portlet_id, renderRequest, renderResponse)
```

PORLET.FTL PORTLET CONFIGURATION

- ❖ This contains the information for the configuration menu (Configuration, Permissions, Look and Feel).
- ❖ The rest of the files contain the HTML markup for the application topper and the application content.
- ❖ It is possible to add CSS classes, change markup, or add custom information.
- ❖ For styling the look and feel of all applications, use the CSS classes found in this file in conjunction with the application decorators to achieve the desired look and feel.

```
portlet_configuration_icons =  
    portlet_configuration_icon_menu.getPortletConfigurationIcons(  
        portlet_display_root_portlet_id, renderRequest, renderResponse)
```

Notes:

Appendix - Taglibs



TAGLIBS: INTRODUCTION

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

INTRODUCTION

- ❖ Taglibs are a tool to create consistent, responsive, accessible UI components for use in development.
- ❖ Each taglib has its own series of attributes that are used to build components.
- ❖ Taglibs are good to use for future-proof development, which means that as the taglibs are updated through new patches and release versions, the taglibs used in development will also be updated.

TAGLIB DOCUMENTATION

- ❖ Documentation for taglibs can be found here:
 - *TLD files in the repo:* <https://github.com/liferay/liferay-portal/tree/master/util-taglib/src/META-INF>
 - *Java docs:* <https://docs.liferay.com/portal/7.0/taglibs/util-taglib/>
- ❖ The documentation informs developers of the attributes that can be used, the type of value that is accepted by the attribute, a description of the attribute, and whether or not the attribute is required.

USING TAGLIBS IN APPLICATIONS

- ❖ To use the various taglibs, make sure you include this in the `init.jsp` of the application.

```
<%@ taglib uri="http://java.sun.com/jsp/jstl/core" prefix="c" %>

<%@ taglib uri="http://java.sun.com/portlet_2_0" prefix="portlet" %>

<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %><%@
taglib uri="http://liferay.com/tld/portlet" prefix="liferay-portlet" %><%@
taglib uri="http://liferay.com/tld/theme" prefix="liferay-theme" %><%@
taglib uri="http://liferay.com/tld/ui" prefix="liferay-ui" %><%@
taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>
```

- ❖ From here, it is possible to call the different taglibs using
`<PREFIX:{tag-name} [TAG-ATTRIBUTES] />`.

Notes:



FORMS AND VALIDATION

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

SETTING UP VALIDATION ON FORMS

- Form, input, and validation tags are provided in Liferay to make creating forms easy and flexible.
- The `<aui:form>` taglib sets up the necessary code (HTML, JS), utilizes Liferay JavaScript APIs, calls the validation framework, and submits the form data to the back-end.
- The “input” taglibs (`<aui:input>`, `<liferay-ui:input-*>`, etc.) generate the necessary code (HTML, CSS, JS) to keep a consistent form UI.
 - These taglibs also initialize the input’s state - e.g., value, disabled, `readonly`.
- The `<aui:validator>` taglib pairs with some input taglibs to provide validation rules for those inputs.
 - This way, you can ensure your users are entering proper data, formatted the way you expect, before it gets sent to the back-end for final validation and processing.

INPUT VALIDATION SUPPORT

- ❖ The inputs that support validators natively are:
 - ❖ <aui:input>
 - ❖ <aui:select>
 - ❖ <liferay-ui:input-date>
 - ❖ <liferay-ui:input-search>

CREATING A FORM

- ❖ Here is an example of a simple form.

```
<aui:form action="<% myActionURL %>" method="post" name="myForm">
    <aui:input label="My Text Input" name="myTextInput" type="text" value='
        <%= "My Text Value" %>' />

    <aui:button type="submit" />
</aui:form>
```

- ❖ Nothing too complex about this, but it ends up outputting all the “magic” that’s needed to pass the data to the back-end.

ADDING VALIDATION

- ❖ What if we want to ensure the user enters the required data? We add validators.

```
<aui:input label="My Text Input" name="myTextInput" type="text" value='
<%= "My Text Value" %>'>
    <aui:validator name="required" />
</aui:input>
```

- ❖ This will force the user to enter something into the input before the form is submitted.

A screenshot of a web form. At the top, there is a text input field labeled "My Text Input *". Below the input field, a red error message says "This field is required.". At the bottom of the form is a blue "Save" button.

ADDING VALIDATION

- ❖ Let's say we wanted to restrict the value to a number between 0 and 10.
- ❖ We can add additional validators. Each one will need to pass in order for the form to submit.

```
<aui:validator name="required" />
<aui:validator name="number" />
<aui:validator name="range">[0,10]</aui:validator>
```

- ❖ We can even customize the error message.

```
<aui:validator errorMessage="Please enter how many fingers you have."
name="range">[0,10]</aui:validator>
```

- ❖ There's a wide variety of built-in validators available. Check the full documentation for details:

https://dev.liferay.com/develop/tutorials/-/knowledge_base/6-2/using-the-alloyui-validator-tag#available-validation-rules

DYNAMICALLY/CONDITIONALLY REQUIRING AN INPUT

- ❖ Sometimes you'll want to validate an input based on the value of another input.
- ❖ You can do this by checking for that condition in a JavaScript function in the required validator's body.

```
<aui:input label="My Checkbox" name="myCheckbox" type="checkbox" />

<aui:input label="My Text Input" name="myTextInput" type="text">
    <aui:validator name="required">
        function() {
            return AUI.$('#<portlet:namespace />myCheckbox').prop('checked');
        }
    </aui:validator>
</aui:input>
```

CUSTOM VALIDATION

- ❖ So far, we've only seen validator rules that come with AUI.
- ❖ Suppose you need something a little more custom or advanced.
- ❖ You can write your own validator and optionally supplement it with built-in validators.

```
<aui:input label="Email" name="email" type="text">
    <aui:validator name="email" />
    <aui:validator errorMessage="Only emails from @example.com are allowed." name="custom">
        function(val, fieldNode, ruleValue) {
            var regex = new RegExp(/@example\.com$/i);
            return regex.test(val);
        }
    </aui:validator>
</aui:input>
```

- ❖ This example runs the regular email validator and your custom domain validator.

ADD ADDITIONAL VALIDATION VIA JAVASCRIPT

- ❖ Sometimes you need to dynamically add additional validation after the page has rendered.
- ❖ Maybe some additional inputs were added to the DOM via an AJAX request.
- ❖ To do this, you'll need to access the `Liferay.Form` object.

```
<aui:script use="liferay-form">
    var form = Liferay.Form.get('<portlet:namespace />myForm');
    // ...
```

- ❖ With the `Liferay.Form` object, you can now `get()` and `set()` `fieldRules`.
- ❖ `fieldRules` are the JavaScript equivalent of all the validators attached to the form.

CUSTOM VALIDATOR EXAMPLE

```
<aui:script use="liferay-form">
    var form = Liferay.Form.get('<portlet:namespace />myForm');
    var oldFieldRules = form.get('fieldRules');
    var newFieldRules = [
        {
            body: function (val, fieldNode, ruleValue) {
                return (val !== '2');
            },
            custom: true,
            errorMessage: 'must-not-equal-2',
            fieldName: 'fooInput',
            validatorName: 'custom_fooInput'
        },
        {
            fieldName: 'fooInput',
            validatorName: 'number'
        }
    ];
    var fieldRules = oldFieldRules.concat(newFieldRules);
    form.set('fieldRules', fieldRules);
</aui:script>
```

MANUAL VALIDATION

- ❖ You may need to execute validation on an input based off of some event not typical of user input or validate a related input at the same time.
- ❖ You can do this by accessing the `formValidator` object and calling the `validateField()` method, passing the input's name.

MANUAL VALIDATION EXAMPLE

```
<aui:input label="Old Title" name="oldTitle" type="text">
    <aui:validator errorMessage="The New Title cannot match the Old Title"
        name="custom">
        function(val, fieldNode, ruleValue) {
            <portlet:namespace />checkOtherTitle('<portlet:namespace />newTitle'
                // check if old and new titles are a match
                return !match;
            }
        </aui:validator>
    </aui:input>
<aui:input label="New Title" name="newTitle" type="text">
    <!-- same custom validator -->
</aui:input>
<aui:script use="liferay-form">
    function <portlet:namespace />checkOtherTitle(fieldName) {
        var formValidator = Liferay.Form.get('<portlet:namespace />myForm').
            formValidator;

        formValidator.validateField(fieldName);
    }
</aui:script>
```

Notes:



TAGLIBS: PAGE LAYOUT

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

AUI TAGLIBS: CONTAINER AND ROW

- Page layout taglibs allow developers to create layouts using Bootstrap 3 in their applications.
- The following `<aui:>` taglibs can be used to create layouts:
 - `<aui:container>`: Creates a container `<div>` tag to wrap `<aui:row>` components and offers additional styling
 - (boolean) `fluid`: Whether to enable the container to span the entire width of the viewport. The default value is `true`.
 - (String) `cssClass`: A CSS class for styling the component
 - (String) `id`: An ID for the component instance
 - `dynamicAttributes` Map: Map of data- attributes for your container
 - `<aui:row>`: Creates a row to hold `<aui:col>` components
 - (String) `cssClass`: A CSS class for styling the component
 - (String) `id`: An ID for the component instance

AUI TAGLIBS: COLUMNS (STRING)

- <aui:col>: Creates a column to display content in an <aui:row> component
 - (String) cssClass: A CSS class for styling the component
 - (String) id: An ID for the component instance
 - (String) lg: Comma-separated string of numbers 1-12 to be used for Bootstrap grid col-lg-
 - (String) md: Comma-separated string of numbers 1-12 to be used for Bootstrap grid col-md-
 - (String) sm: Comma-separated string of numbers 1-12 to be used for Bootstrap grid col-sm-
 - (String) xs: Comma-separated string of numbers 1-12 to be used for Bootstrap grid col-xs-

AUI TAGLIBS: COLUMNS (INT)

- (int) span: The width of the column in the containing row as a fraction of 12. For example, a span of 4 would result in a column width of 4/12 (or 1/3) of the total width of the containing row.
- (int) width: The width of the column in the containing row as a percentage, overriding the span attribute. The width is then converted to a span expressed as ((width/100) x 12), rounded to the nearest whole number. For example, a width of 33 would be converted to 3.96, which would be rounded up to a span value of 4.

EXAMPLE JSP

- ❖ Here is an example layout created in an application.

```
<%@ taglib uri="http://liferay.com/tld/aui" prefix="aui" %>
<aui:container cssClass='super-awesome-container'>
    <aui:row>
        <aui:col width="<%=" 40 %>">
            <h2>Some fun content using the 'width' attribute</h2>
        </aui:col>

        <aui:col width="<%=" 60 %>">
            <p>
                Swag mollit waistcoat biodiesel nesciunt.
            </p>
        </aui:col>
    </aui:row>
</aui:container>
```

Notes:



TAGLIBS: UI COMPONENTS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

COMPONENTS

- ❖ There are a lot of components that can be created using the `<liferay:ui>` taglibs.
- ❖ The benefit of using these is the markup will be consistent, responsive, and accessible across your applications.
- ❖ The markup generated when using the taglibs can be found on the *Liferay Github Repo*: <https://github.com/liferay/liferay-portal/tree/master/portal-web/docroot/html/taglib/ui>

USING LIFERAY-UI: TAGLIBS

- ❖ A list of the available <liferay-ui> taglibs can be found here:
<https://docs.liferay.com/portal/7.0/taglibs/util-taglib/>
- ❖ To use the taglib library, you'll need to make sure the following line is in your JSP:

```
<%@ taglib prefix="liferay-ui" uri="http://liferay.com/tld/ui" %>
```
- ❖ Each taglib has a list of attributes that can be passed to the tag.
- ❖ Some of these are required and some are optional. Looking at the documentation can help you find which ones you'll need.

EXAMPLES OF LIFERAY-UI: TAGLIBS

- ❖ Here is an example of the <liferay-ui:alert> taglib. Using the example below will create a success alert that the user can close:

```
<liferay-ui:alert  
    closeable="true"  
    icon="exclamation-full"  
    message="Here is our awesome alert example"  
    type="success"  
/>
```

- ❖ Here is an example of a <liferay-ui:user-display>:

```
<liferay-ui:user-display  
    markupView="lexicon"  
    showUserDetails="true"  
    showUserName="true"  
    userId="<%=\n        themeDisplay.getRealUserId()\n    %>"  
    userName="<%=\n        themeDisplay.getRealUser().getFullName()\n    %>"  
/>
```

Notes:



TAGLIBS: UTILITY COMPONENTS

Copyright ©2017 Liferay, Inc.

All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

USING LIFERAY-UTIL: TAGLIBS

- ❖ The <liferay-util> taglib is used to pull in other resources into an application or theme.
- ❖ It can also be used to dictate which resources need to be inserted at the bottom or top of the HTML source.
- ❖ A list of the available <liferay-util> taglibs can be found at <https://docs.liferay.com/portal/7.0/taglibs/util-taglib/>
- ❖ To use the taglib library, you'll need to make sure the following line is in your JSP:
`<%@ taglib prefix="liferay-util" uri="http://liferay.com/tld/util" %>`
- ❖ Each taglib has a list of attributes that can be passed to the tag.
- ❖ Since each of the <liferay-util> taglibs is unique, we'll go over each one briefly.

USING LIFERAY-UTIL: BODY-BOTTOM

- The content placed between the opening and closing of this tag will be moved to the bottom of the body tag.
- This tag allows you to insert your own HTML markup at the bottom of the body tag for the page.
- The attribute outputKey is the reference key for this content.

EXAMPLE OF LIFERAY-UTIL: BODY-BOTTOM

- Here is an example of using <liferay-util:body-bottom>:

```
<liferay-util:body-bottom outputKey="productMenu">
    <div class="lfr-product-menu-panel sidenav-fixed sidenav-menu-slider" id=""
        <%= portletNamespace %>sidenavSliderId">
        <div class="product-menu sidebar sidenav-menu">
            <liferay-portlet:runtime portletName="<%= ProductNavigationProductMenuPortletKeys.
                PRODUCT_NAVIGATION_PRODUCT_MENU %>" />
        </div>
    </div>
</liferay-util:body-bottom>
```

USING LIFERAY-UTIL: BODY-TOP

- ❖ The content placed between the opening and closing of this tag will be moved to the top of the body tag.
- ❖ This tag allows you to insert your own HTML markup at the top of the body tag for the page.
- ❖ The attribute outputKey is the reference key for this content.
- ❖ Here is an example of using <liferay-util:body-top>:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:body-top outputKey="topContent">
    <div>
        <h1>I'm at the top of the page!</h1>
    </div>
</liferay-util:body-top>
```

USING LIFERAY-UTIL: BUFFER

- ❖ The content placed between the opening and closing of this tag is saved to the value of the var attribute.
- ❖ This allows you to build a piece of markup that can be reused in a JSP.

Here is an example of using <liferay-util:buffer>:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>
<liferay-util:buffer var="myBuffer">
    <small class="text-capitalize text-muted">
        This is my buffer content
    </small>
</liferay-util:buffer>
<div class="container">
    <h1>Welcome!</h1>
    <%= myBuffer %>
</div>
<div class="container">
    <h1>A Wonderful Title!</h1>
    <%= myBuffer %>
</div>
```

USING LIFERAY-UTIL: DYNAMIC-INCLUDE

- ❖ This taglib allows you to register some content with the DynamicIncludeRegistry.
- ❖ Read more about the OSGi Service Registry here: <http://docs.spring.io/osgi/docs/current/reference/html/service-registry.html>
- ❖ The Service Registry makes it easier for modules in the registry to use the content included from the taglib.
- ❖ Here is an example:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:dynamic-include key="/path/to/jsp#pre" />

<div>
    <p>And here we have our content</p>
</div>

<liferay-util:dynamic-include key="/path/to/jsp#post" />
```

USING LIFERAY-UTIL: GET-URL

- ❖ This tag scraps the URL provided by the url attribute.
- ❖ If a value is provided for the var attribute, the content from the screen-cap is scoped to that variable.
- ❖ Otherwise, the content will be displayed where the taglib is used.

EXAMPLE OF LIFERAY-UTIL: GET-URL

- ❖ Here is an example:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:get-url url="https://www.google.com/" />

<!-- Using var attribute -->

<liferay-util:get-url url="https://www.google.com/" var="google" />

<div>
    <h2>We stole <a href="https://www.google.com/">Google</a>, here it is.</h2>

    <div class="google">
        <%= google %>
    </div>
</div>
```

USING LIFERAY-UTIL: HTML-BOTTOM

- ❖ The content placed between the opening and closing of this tag will be moved to the bottom of the html tag.
- ❖ This tag allows you to insert your own HTML markup at the bottom of the body tag for the page.
- ❖ The attribute outputKey is the reference key for this content.

EXAMPLE OF LIFERAY-UTIL: HTML-BOTTOM

- ❖ Here is an example of using <liferay-util:html-bottom>. Many times, the content passed to this taglib will be JavaScript:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:html-bottom outputKey="taglib_alert_user">
    <aui:script use="liferay-alert">
        new Liferay.Alert(
            {
                closeable: true,
                message: 'Just saying hello from the &lt;liferay-util:html-bot
                type: 'success'
            }
        ).render(#wrapper);
    </aui:script>
</liferay-util:html-bottom>
```

USING LIFERAY-UTIL:HTML-TOP

- ❖ The content placed between the opening and closing of this tag will be moved to the head tag.
- ❖ This tag allows you to insert your own HTML markup at the top of the body tag for the page. The attribute outputKey is the reference key for this content.

- ❖ Here is an example of using <liferay-util:html-top>:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:html-top>
    <link data-senna-track="permanent" href="/path/to/style.css"
          rel="stylesheet" type="text/css" />
</liferay-util:html-top>
```

USING LIFERAY-UTIL: INCLUDE

- ❖ This tag can be used to include other JSP files in an application.
- ❖ It can help for readability in an application and a separation of concerns for JSP files.
- ❖ The page attribute is required and the value is the path to the JSP or JSF to be included. The servletContext attribute refers to the request context that the included JSP should use.
- ❖ By passing `<%= application %>` to this attribute, the included JSP can use the same request object and other objects that might be set in the prior JSP.
- ❖ Here is an example of using `<liferay-util:include>`:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:include page="/path/to/view.jsp" servletContext=
<%= application %> />
```

USING LIFERAY-UTIL: PARAM

- ❖ This tag can be used to add a parameter value to a url.
- ❖ It is useful when used in tandem with `<liferay-util:include>` for accessing new parameter values in another JSP.
- ❖ Here is an example of using `<liferay-util:param>`:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:include page="/path/to/answer.jsp" servletContext=
<%= application %>">
    <liferay-util:param name="answer" value="42" />
</liferay-util:include>
```

- ❖ In `answer.jsp`:

```
<%@ page import="com.liferay.portal.kernel.util.ParamUtil" %>

<div>
    <p>The answer to life the universe and everything is <%= ParamUtil.getString
        (request, "answer") %></p>
</div>
```

USING LIFERAY-UTIL: WHITESPACE-REMOVER

- ❖ This taglib is used for removing all whitespace from whatever is included between the opening and closing of the tag.
- ❖ Here is an example of using <liferay-util:whitespace-remover>:

```
<%@ taglib uri="http://liferay.com/tld/util" prefix="liferay-util" %>

<liferay-util:whitespace-remover>
    <div class="nput-container">
        <label for="myInput">
            Is the &lt;liferay-util:whitespace-remover&gt; taglib fantastic!
        </label>

        <input class="input" id="myInput" name="myInput" type="checkbox">
    </div>
</liferay-util:whitespace-remover>
```

Notes:

Appendix - Upgrading a Theme to DXP



UPGRADING AN EXISTING 6.2 THEME TO DXP

Copyright ©2017 Liferay, Inc.
All Rights Reserved.

No material may be reproduced electronically or in print,
distributed, copied, sold, resold, or otherwise exploited
for any commercial purpose without express written
consent of Liferay, Inc.

MIGRATING 6.2 THEMES

- ❖ In DXP, upgrading themes from 6.2 is made much easier.
- ❖ Using the Liferay Theme Generator, developers can import and upgrade themes that were originally created in the 6.2 plugins SDK.
- ❖ Let's take a look at what a developer needs to do to upgrade an existing theme.

IMPORTING WITH THE LIFERAY THEME GENERATOR

- ❖ Developers need to import their theme first before upgrading.
- ❖ Importing includes the following steps:
 1. First, you need to navigate to the 6.2 theme directory and run the following:
`yo liferay-theme:import`
 2. Next, you'll need to enter the path to the theme and hit enter.
 - The theme's modified files are all copied and migrated to a newly created `src` directory.
 - A `gulpfile.js`, `liferay-theme.json`, `package.json` file and a `node_modules` directory is added to the existing theme.
 3. Finally, you'll need to enter the path to your app server as well as the URL.
- ❖ The theme can now use gulp tasks, but needs a few upgrades to function on DXP.

GULP UPGRADE

- ❖ With the theme set to use gulp tasks, developers can take advantage of the upgrade tasks.
 1. In the theme's root directory, you can run the following:
`gulp upgrade`
 - The theme's files are placed in a `_backup` folder.
 - If you want to revert to the 6.2 theme, you can use `gulp upgrade:revert`.
 2. Hitting Enter will rename all the existing `.css` files to `.scss`.
 - All Sass files now use the `.scss` extension and Sass partials are indicated with a `_` at the beginning of the file name.
- ❖ The upgrade tasks will check all the theme's files and either upgrade or leave suggestions.
- ❖ To comply with DXP, the theme's Bootstrap code is also upgraded from version 2 to version 3.

USING FONT AWESOME

- If Font Awesome was in use in the previous theme, developers will need to put the following variables in the `_aui_variables.scss`:

```
// Icon paths
$FontAwesomePath: "aui/lexicon/fonts/alloy-font-awesome/font";
$font-awesome-path: "aui/lexicon/fonts/alloy-font-awesome/font";
$icon-font-path: "aui/lexicon/fonts/";
```

UPDATING THE RESPONSIVE CSS

- In DXP, the `respond-to` mixins have been replaced with explicit media queries.
- Developers will need to update their responsive CSS to the following:

1. `@include respond-to(phone) : @include media-query(null, $screen-xs-max)`
2. `@include respond-to(tablet) : @include media-query(sm, $screen-sm-max)`
3. `@include respond-to(phone, tablet) : @include media-query(null, $breakpoint_tablet - 1)`
4. `@include respond-to(desktop, tablet) : @include sm`
5. `@include respond-to(desktop) : @include media-query($breakpoint_tablet, null)`

THE RESOURCES IMPORTER

- The Resources Importer has undergone several structural changes.
- This will impact both the configuration files and the directory structure.
- Let's look at what we need to update to make it all come together on DXP.

UPDATING THE PLUGIN PACKAGE PROPERTIES FILE

- In the `liferay-plugin-package.properties` file, the following line should be removed:
`required-deployment-contexts`
 - This property is no longer required, as the Resources Importer is now an OSGi module that is included in Liferay.
- The value of the `resources-importer-target-class-name` property needs to be updated to the following:
`com.liferay.portal.kernel.model.Group`

RESOURCES IMPORTER CONTENT XML

- ❖ In previous version of Liferay, web content was included in the Resources Importer by including `html` files.
- ❖ In DXP, all articles need to include a structure, template, and an `xml` file instead of `html`.
- ❖ Upgrading articles can easily be done by renaming the files to `[article-name].xml` and using the following pattern:

```
<?xml version="1.0"?>
<root available-locales="en_US" default-locale="en_US">
    <dynamic-element name="content" type="text_area"
        index-type="keyword" index="0">
        <dynamic-content language-id="en_US">
            <![CDATA[
                HTML CONTENT GOES HERE
            ]]>
        </dynamic-content>
    </dynamic-element>
</root>
```

STRUCTURES: FROM XML TO JSON

- ❖ In DXP, structures need to be JSON files.
- ❖ Developers can find the JSON source of any upgraded structures in *Menu→Site Administration→Content→Structures*.
- ❖ All Structure files need to be created as `[structure-name].json` and go in the `resources-importer/journal/structures/` directory.
- ❖ As mentioned above, every content article needs a structure and template.
- ❖ Let's look at the basic content structure below.

BASIC WEB CONTENT STRUCTURE: TOP

```
{  
    "availableLanguageIds": [  
        "en_US"  
    ],  
    "defaultLanguageId": "en_US",  
    "fields": [  
        {  
            "label": {  
                "en_US": "Content"  
            },  
            "predefinedValue": {  
                "en_US": ""  
            },  
            "style": {  
                "en_US": ""  
            },  
            "tip": {  
                "en_US": ""  
            },  
            ...  
        }  
    ]  
}
```

BASIC WEB CONTENT STRUCTURE: BOTTOM

```
...  
    "dataType": "html",  
    "fieldNamespace": "ddm",  
    "indexType": "keyword",  
    "localizable": true,  
    "name": "content",  
    "readOnly": false,  
    "repeatable": false,  
    "required": false,  
    "showLabel": true,  
    "type": "ddm-text-html"  
}  
]  
}
```

- ❖ The structure above defines some basic attributes for the web content, sets the input field data as html, and identifies the web content by the ``name``: ``content``.

CONTENT TEMPLATES

- ❖ Developers can place their existing .ftl templates in the resources-importer/journal/templates folder.
- ❖ This will also work with any .vm templates.
- ❖ At the very least, there must be a basic Web Content Template with the following:
 `${content.getData()}`
- ❖ And with that, your theme is ready to deploy on DXP!

Notes: