

Introduction

Getting Started

Framework Concepts

Components

Understanding Infusion Components

Understanding Component Options and their Defaults

Component Grades

Options Merging

Component Lifecycle

Inversion of Control

How to use Infusion IoC

Subcomponent Declaration

Contexts

Invokers

Expansion of Component Options

IoCSS

Declarative this-ism in IoC

IoC References

Events

Infusion Event System

Event injection and boiling

ChangeApplier

Model Relay

Progressive Enhancement

Renderer

How To Use the Renderer

Component Trees

Renderer Component Trees

ProtoComponent Types

Renderer Component Tree Expanders

Cutpoints

Renderer Components

Renderer Decorators

Preferences Framework

Preferences Editor

Builder

Primary Schema

Auxiliary Schema

Enactors

Understanding Infusion Components

[Edit on Github](#)

***Note:** This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the APIs may change.*

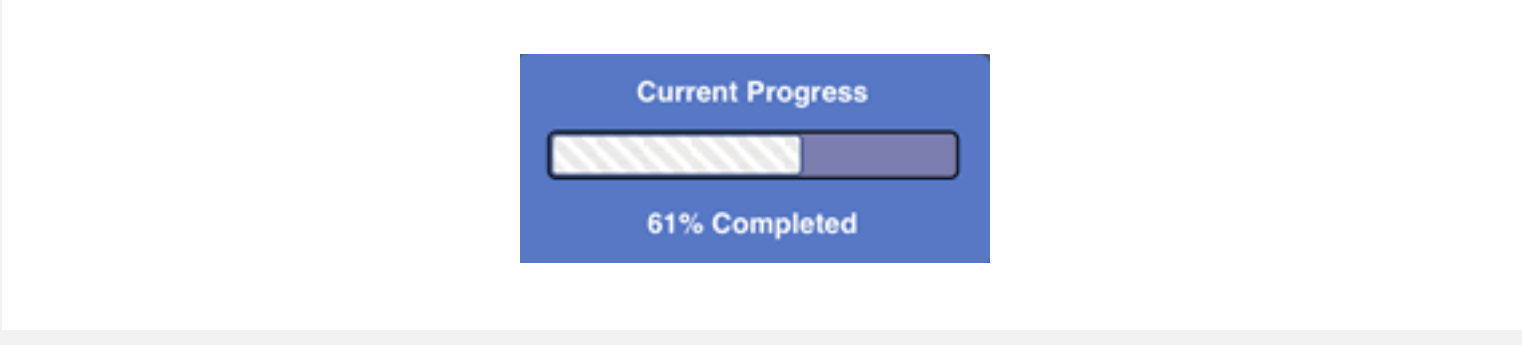
The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

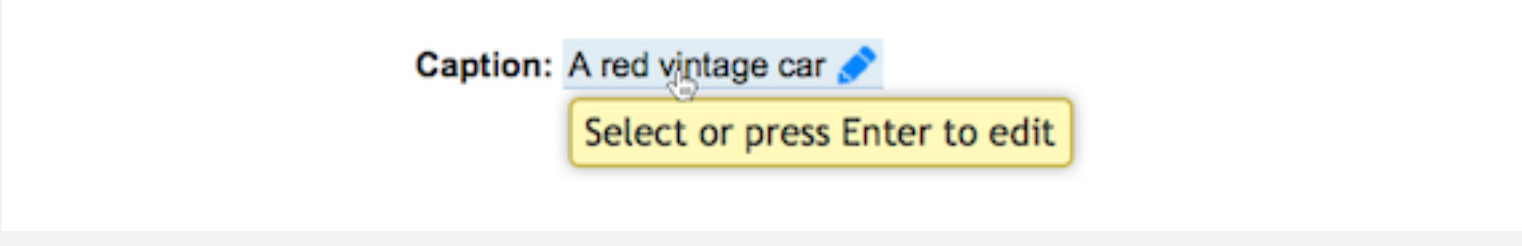
To help understand how a widget or application might be designed using components, consider some of the components in the Infusion Component Library:

Progress



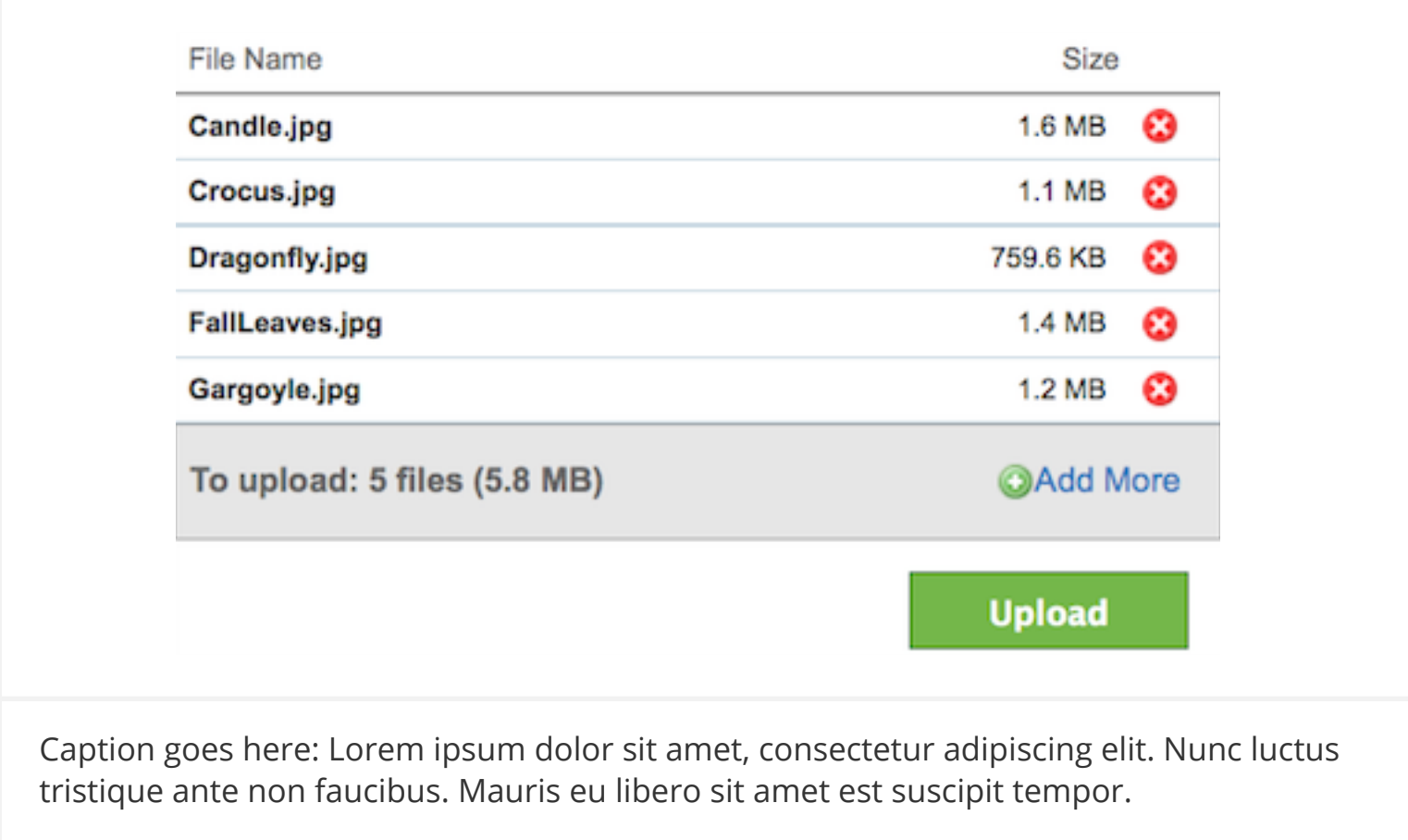
The Infusion Progress component is single component with no subcomponents. It has a number of UI elements that work together and are updated programmatically to show the progress of some activity. It has a pretty simple purpose and function, one that doesn't make much sense to try to chunk up into multiple components.

Inline Edit



The Inline Edit component allows user to edit text in place, without switching to a new screen, by simply switching into an in-place edit mode. The view mode is implemented one way, with certain functionality (i.e. a tooltip, an affordance to edit), and the edit mode is implemented differently: it's an edit field. Conceptually, these two modes are rather different, and so they're implemented as two separate subcomponents of the main Inline Edit component.

Uploader



The Uploader allows users to add several files to a queue and then upload them all at once. It is actually made up of several subcomponents: It has the file queue view, which displays the files currently in the queue; it has a total progress bar at the bottom. In turn, the file queue view component has its own subcomponents.

What Does A Component Look Like?

A component is a regular JavaScript object that has certain characteristics. The most simple components have a typeName and an id, but typical components will have more:

Most will have:

- **a creator function**
 - the function that implementors invoke, which returns the component object itself
- **configuration options**
 - various values that control the operation of the component, which can be overridden by implementors to customize the component
- **public functions**

Depending on what the component is for, some will include infrastructure to support

- events
- a model
- a view
- a renderer

New kinds of components are created by passing configuration information to the 'fluid.defaults' function. This function will create the **creator function** that will be used to instantiate the component. The Framework provides [supports for automatically creating components of various types, or 'grades'](#); as well, developers can create their own grades.

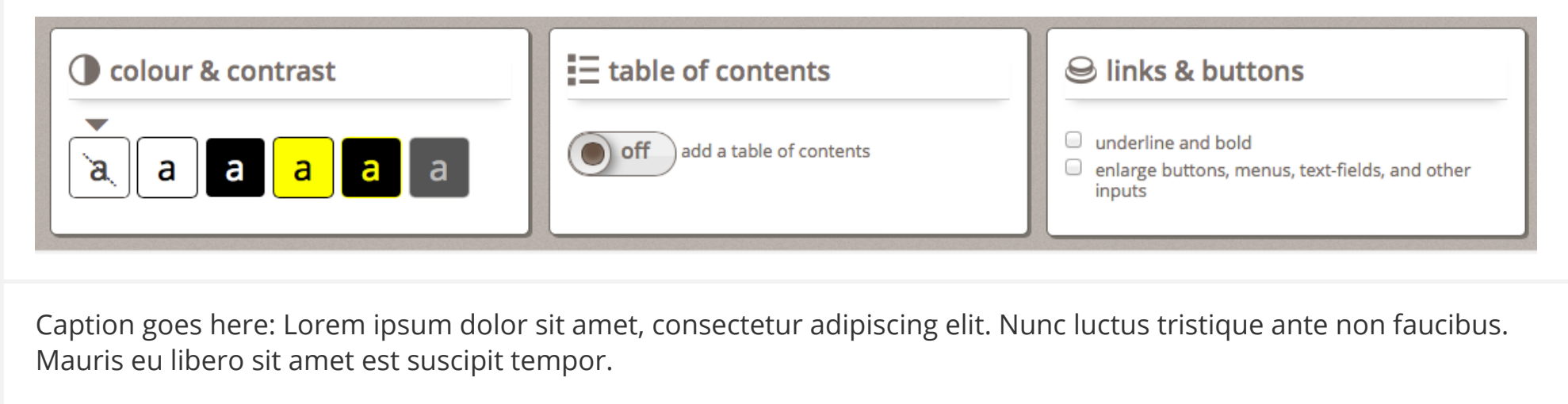
Examples with code

```
fluid.enhance.check({
  check1: "my.checking.function1",
  check2: "my.checking.function2",
  ...
});
```

The function `fluid.enhance.check()` will execute the specified functions and store the results in the static environment using the associated key (e.g. `check1`). The presence of the tags in the static environment can be used in the context argument to `fluid.demands()` .

Decorator Type	Field Name	Field Type	Field Description	Example
Decorator or Type	func	String	jQuery function to be invoked	decorators: [{ type: "jQuery", func: "click", args: function() \${this}.hide(); }]
	args	Array of Object	Arguments to the jQuery function	

Tables and larger images can fill all available width if required.



Caption goes here: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc luctus tristique ante non faucibus. Mauris eu libero sit amet est suscipit tempor.

Understanding Infusion Components

[Edit on Github](#)

***Note:** This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the APIs may change.*

The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

To help understand how a widget or application might be designed using components, consider some of the components in the Infusion Component Library:

Uploader

File Name	Size
Candle.jpg	1.6 MB
Crocus.jpg	1.1 MB
Dragonfly.jpg	759.6 KB
FallLeaves.jpg	1.4 MB
Gargoyle.jpg	1.2 MB
To upload: 5 files (5.8 MB) Add More	

Upload

Caption goes here: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc luctus tristique ante non faucibus. Mauris eu libero sit amet est suscipit tempor.

The Uploader allows users to add several files to a queue and then upload them all at once. It is actually made up of several subcomponents: It has the file queue view, which displays the files currently in the queue; it has a total progress bar at the bottom. In turn, the file queue view component has its own subcomponents.

What Does A Component Look Like?

A component is a regular JavaScript object that has certain characteristics. The most simple components have a `typeName` and an `id`, but typical components will have more:

- Most will have:
- **a creator function**
 - the function that implementors invoke, which returns the component object itself
 - **configuration options**
 - various values that control the operation of the component, which can be overridden by implementors to customize the component
 - **public functions**

Depending on what the component is for, some will include infrastructure to support

- events
- a model
- a view
- a renderer

New kinds of components are created by passing configuration information to the `'fluid.defaults'` function. This function will create the **creator function** that will be used to instantiate the component. The Framework provides [supports for automatically creating components of various types, or 'grades'](#); as well, developers can create their own grades.

Examples with code

```
fluid.enhance.check({
  check1: "my.checking.function1",
  check2: "my.checking.function2",
  ...
});
```

The function `fluid.enhance.check()` will execute the specified functions and store the results in the static environment using the associated key (e.g. `check1`). The presence of the tags in the static environment can be used in the context argument to `fluid.demands()`.

Decorator Type

Decorator

 or

Type

Field Name

func

Field Type

String

Field Description

jQuery function to be invoked

Example

decorators: [{ type: "jQuery", func: "click", args: function() {\$(this).hide();} }]

Decorator Type

Decorator

 or

Type

Field Name

func

Field Type

Array of Object

Field Description

jQuery function to be invoked

Example

decorators: [{ type: "jQuery", func: "click", args: function() {\$(this).hide();} }]

INFUSION

TUTORIALS

API

Introduction

[Getting Started](#)

[Framework Concepts](#)

Components

[Understanding Infusion Components](#)

[Understanding Component Options and their Defaults](#)

[Component Grades](#)

[Options Merging](#)

[Component Lifecycle](#)

Inversion of Control

[How to use Infusion IoC](#)

[Subcomponent Declaration](#)


[Contexts](#)

[Invokers](#)

[Expansion of Component Options](#)

[What's Fluid project?](#)

Understanding Infusion Components

 [Edit on Github](#)

Note: This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the APIs may change.

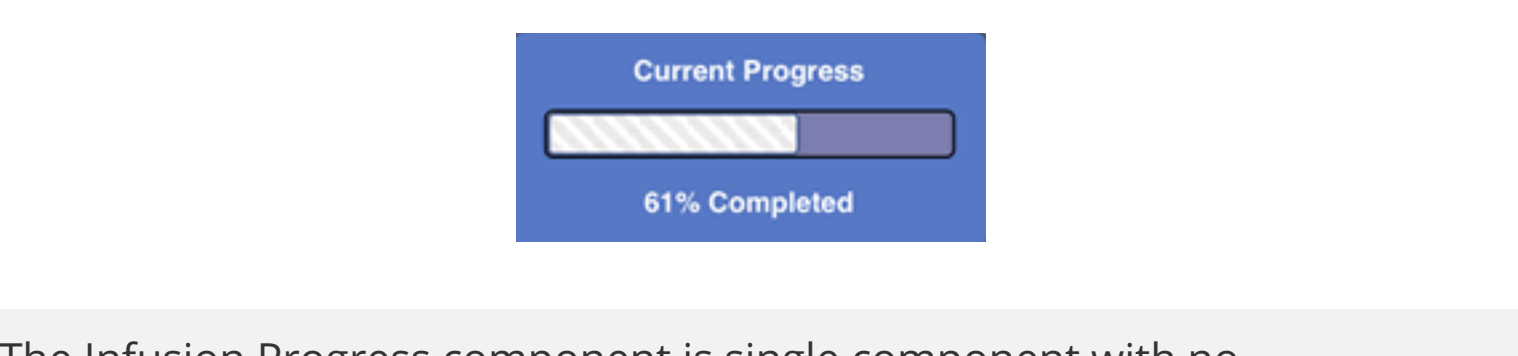
The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

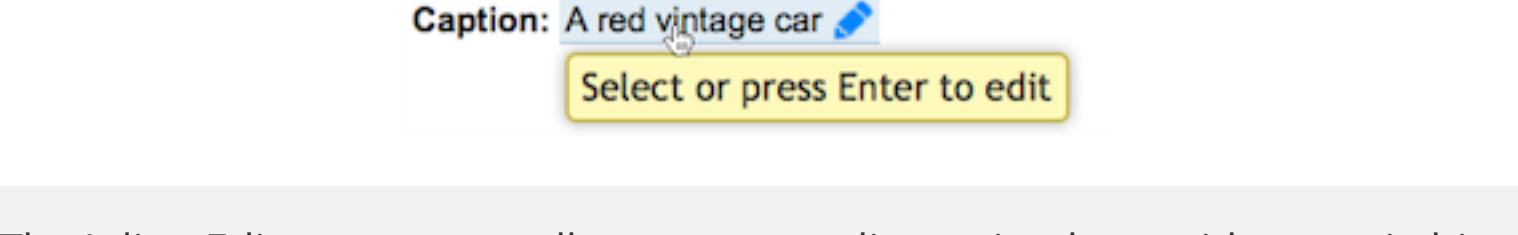
To help understand how a widget or application might be designed using components, consider some of the components in the Infusion Component Library:

Progress



The Infusion Progress component is single component with no subcomponents. It has a number of UI elements that work together and are updated programmatically to show the progress of some activity. It has a pretty simple purpose and function, one that doesn't make much sense to try to chunk up into multiple components.

Inline Edit



The Inline Edit component allows user to edit text in place, without switching to a new screen, by simply switching into an in-place edit mode. The view mode is implemented one way, with certain functionality (i.e. a tooltip, an affordance to edit), and the edit mode is implemented differently: it's an edit field. Conceptually, these two modes are rather different, and so they're implemented as two separate subcomponents of the main Inline Edit component.

Uploader



The Uploader allows users to add several files to a queue and then upload them all at once. It is actually made up of several subcomponents: It has the file queue view, which displays the files currently in the queue; it has a total progress bar at the bottom. In turn, the file queue view component has its own subcomponents.

What Does A Component Look Like?

A component is a regular JavaScript object that has certain characteristics. The most simple components have a typeName and an id, but typical components will have more:

Most will have:

- **a creator function**
 - the function that implementors invoke, which returns the component object itself
- **configuration options**
 - various values that control the operation of the component, which can be overridden by implementors to customize the component
- **public functions**

Depending on what the component is for, some will include infrastructure to support

- events
- a model
- a view
- a renderer

New kinds of components are created by passing configuration information to the 'fluid.defaults' function. This function will create the **creator function** that will be used to instantiate the component. The Framework provides [supports for automatically creating components of various types, or 'grades'](#); as well, developers can create their own grades.

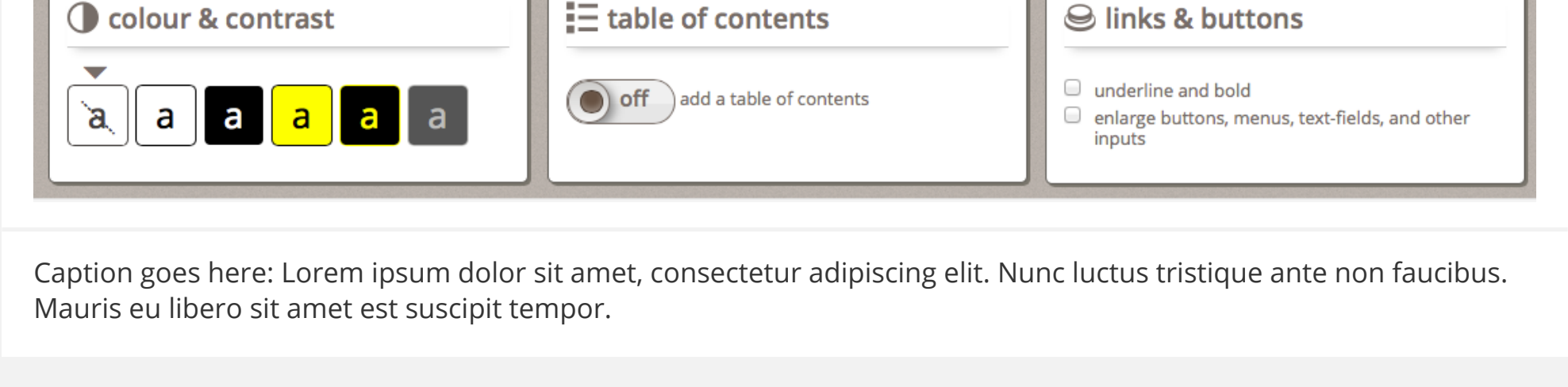
Examples with code

```
fluid.enhance.check({
  check1: "my.checking.function1",
  check2: "my.checking.function2",
  ...
});
```

The function `fluid.enhance.check()` will execute the specified functions and store the results in the static environment using the associated key (e.g. `check1`). The presence of the tags in the static environment can be used in the context argument to `fluid.demands()`.

Decorator Type	Field Name	Field Type	Field Description	Example
Decorator or Type	func	String	jQuery function to be invoked	decorators: [{ type: "jQuery", func: "click", args: function() {\$(this).hide();} }]
	args	Array of Object	Arguments to the jQuery function	

Tables and larger images can fill all available width if required.



TUTORIALS

INFUSION

API

Introduction

Getting Started

Framework Concepts

Components

Understanding Infusion Components

Understanding Component Options and their Defaults

Component Grades

Options Merging

Component Lifecycle

Inversion of Control

How to use Infusion IoC

Subcomponent Declaration

Contexts

Invokers

Expansion of Component Options

IoCSS

Declarative this-ism in IoC

IoC References

Events

Infusion Event System

Event injection and boiling

ChangeApplier

Model Relay

Progressive Enhancement

Renderer

How To Use the Renderer

Component Trees

Renderer Component Trees

ProtoComponent Types

Renderer Component Tree Expanders

Cutpoints

Renderer Components

Renderer Decorators

Preferences Framework

Preferences Editor

Builder

Primary Schema

Auxiliary Schema

Enactors

Introduction

Getting Started

Framework Concepts

Components

Understanding Infusion Components

Understanding Component Options and their Defaults

Component Grades

Options Merging

Component Lifecycle

Inversion of Control

How to use Infusion IoC

Subcomponent Declaration

Contexts

Invokers

Expansion of Component Options

IoCSS

Declarative this-ism in IoC

IoC References

Events

Infusion Event System

Event injection and boiling

ChangeApplier

Model Relay

Progressive Enhancement

Renderer

How To Use the Renderer

Component Trees

Understanding Infusion Components

 [Edit on Github](#)

***Note:** This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the **APIs may change**.*

The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

To help understand how a widget or application might be designed using components

Introduction

Getting Started

Framework Concepts

Components

Understanding Infusion Components

Understanding Component Options and their Defaults

Component Grades

Options Merging

Component Lifecycle

Inversion of Control

How to use Infusion IoC

Subcomponent Declaration

Contexts

Invokers

Expansion of Component Options

IoCSS

Declarative this-ism in IoC

IoC References

Events

Infusion Event System

Event injection and boiling

ChangeApplier

Model Relay

Progressive Enhancement

Renderer

How To Use the Renderer

Component Trees

Understanding Infusion Components

 [Edit on Github](#)

***Note:** This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the *APIs may change*.*

The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

To help understand how a widget or application might be designed using components, consider some of the components in the



Introduction

- Getting Started
- Framework Concepts

Components

- Understanding Infusion Components
- Understanding Component Options and their Defaults
- Component Grades
- Options Merging
- Component Lifecycle

Inversion of Control

- How to use Infusion IoC
- Subcomponent Declaration
- Contexts
- Invokers
- Expansion of Component Options
- IoCSS
- Declarative this-ism in IoC
- IoC References

Events

- Infusion Event System
- Event injection and boiling

ChangeApplier

Model Relay

Progressive Enhancement

Renderer

- How To Use the Renderer
- Component Trees

Renderer Component Trees

ProtoComponent Types

Renderer Component Tree Expanders
- Cutpoints
- Renderer Components
- Renderer Decorators

Preferences Framework

- Preferences Editor
- Builder

Primary Schema

Auxiliary Schema
- Enactors

Understanding Infusion Components

Edit on Github

Note: This is an example of a paragraph with emphasis. This functionality is **Sneak Peek** status. This means that the APIs may change.

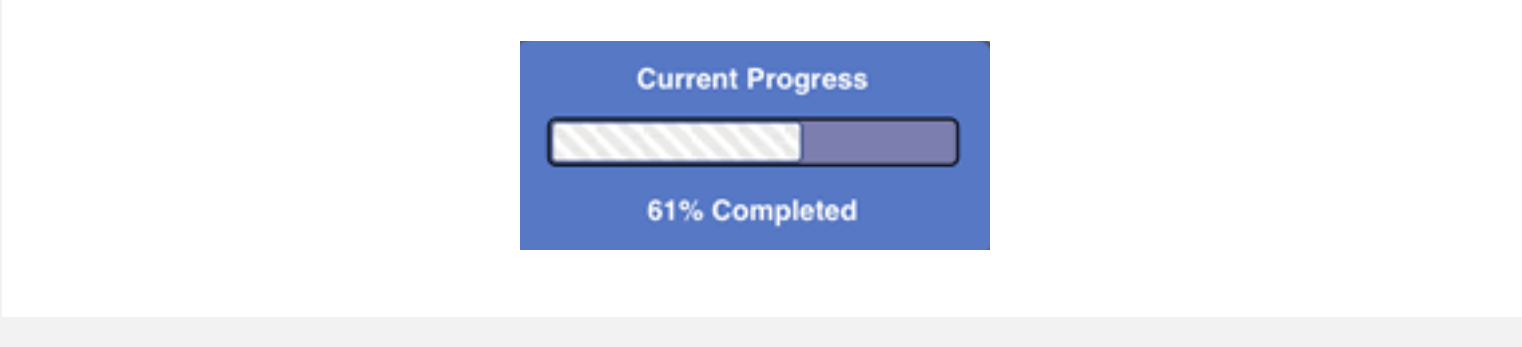
The first paragraph in an article is styled differently. Every Infusion application is structured as a set of **components**. An Infusion component can represent a visible component on screen, a collection of related functionality such as an **object** as in object-orientation, or simply a unit of work or relationship between other components. This page provides resources to help you understand components.

If you're creating an entire web application, your application would be implemented as a component that coordinates interactions between other components that handle the different parts of your application.

Examples

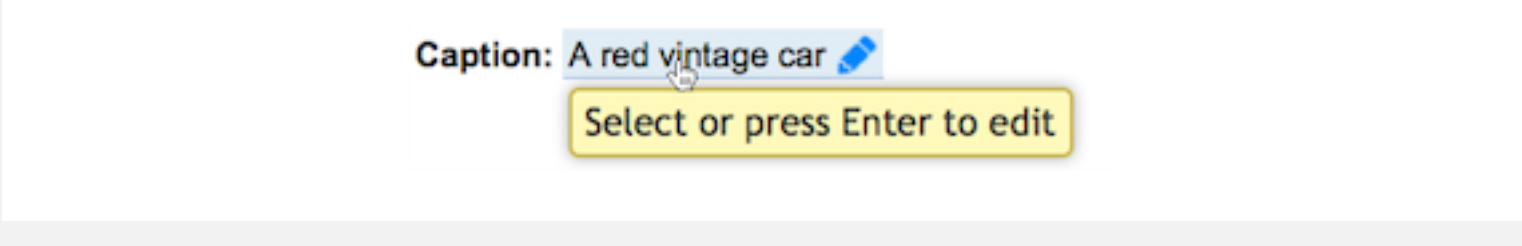
To help understand how a widget or application might be designed using components, consider some of the components in the Infusion Component Library:

Progress



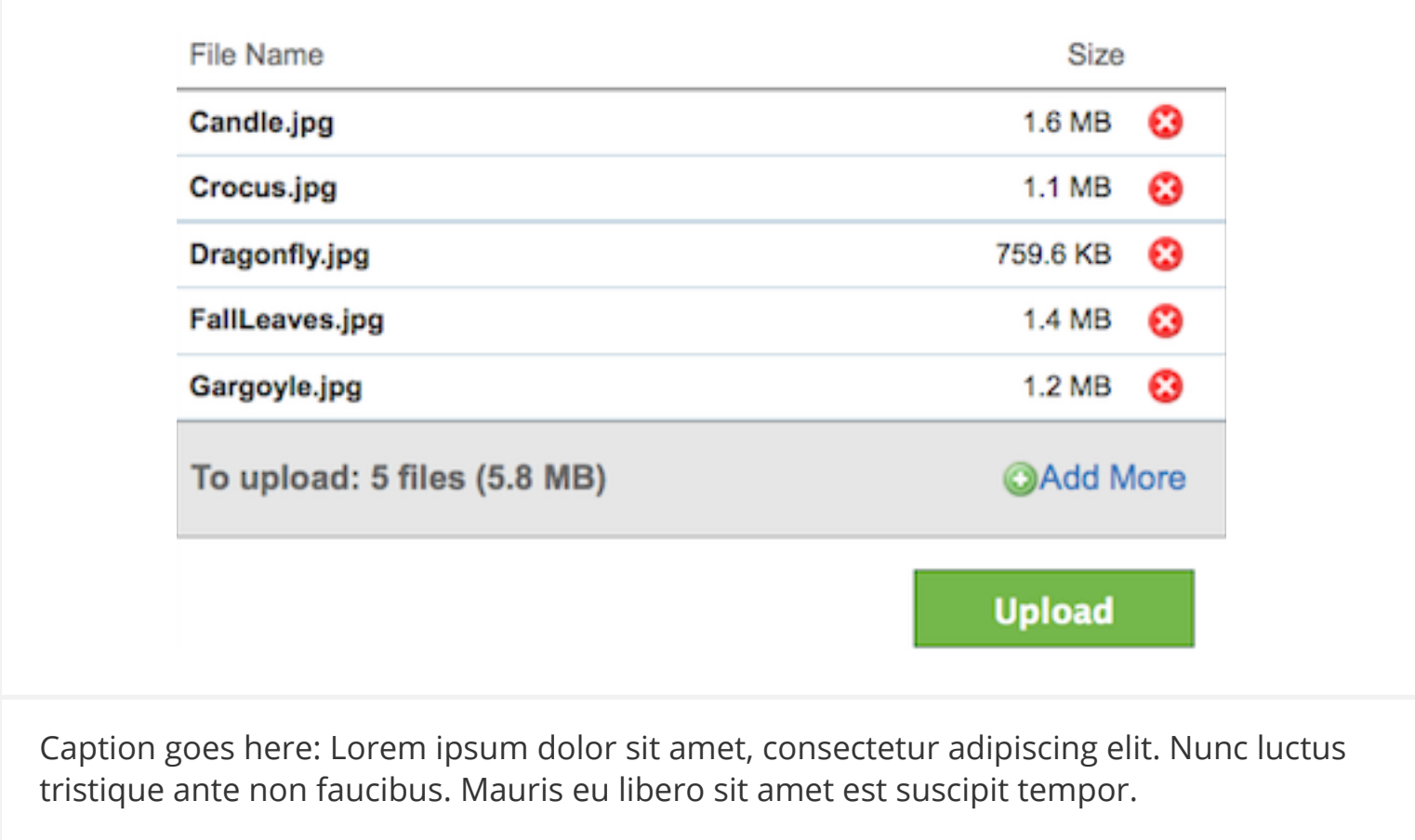
The Infusion Progress component is single component with no subcomponents. It has a number of UI elements that work together and are updated programmatically to show the progress of some activity. It has a pretty simple purpose and function, one that doesn't make much sense to try to chunk up into multiple components.

Inline Edit



The Inline Edit component allows user to edit text in place, without switching to a new screen, by simply switching into an in-place edit mode. The view mode is implemented one way, with certain functionality (i.e. a tooltip, an affordance to edit), and the edit mode is implemented differently: it's an edit field. Conceptually, these two modes are rather different, and so they're implemented as two separate subcomponents of the main Inline Edit component.

Uploader



Caption goes here: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc luctus tristique ante non faucibus. Mauris eu libero sit amet est suscipit tempor.

The Uploader allows users to add several files to a queue and then upload them all at once. It is actually made up of several subcomponents: It has the file queue view, which displays the files currently in the queue; it has a total progress bar at the bottom. In turn, the file queue view component has its own subcomponents.

What Does A Component Look Like?

A component is a regular JavaScript object that has certain characteristics. The most simple components have a typeName and an id, but typical components will have more:

Most will have:

- **a creator function**
 - the function that implementors invoke, which returns the component object itself
- **configuration options**
 - various values that control the operation of the component, which can be overridden by implementors to customize the component
- **public functions**

Depending on what the component is for, some will include infrastructure to support

- events
- a model
- a view
- a renderer

New kinds of components are created by passing configuration information to the 'fluid.defaults' function. This function will create the **creator function** that will be used to instantiate the component. The Framework provides [supports for automatically creating components of various types, or 'grades'](#); as well, developers can create their own grades.

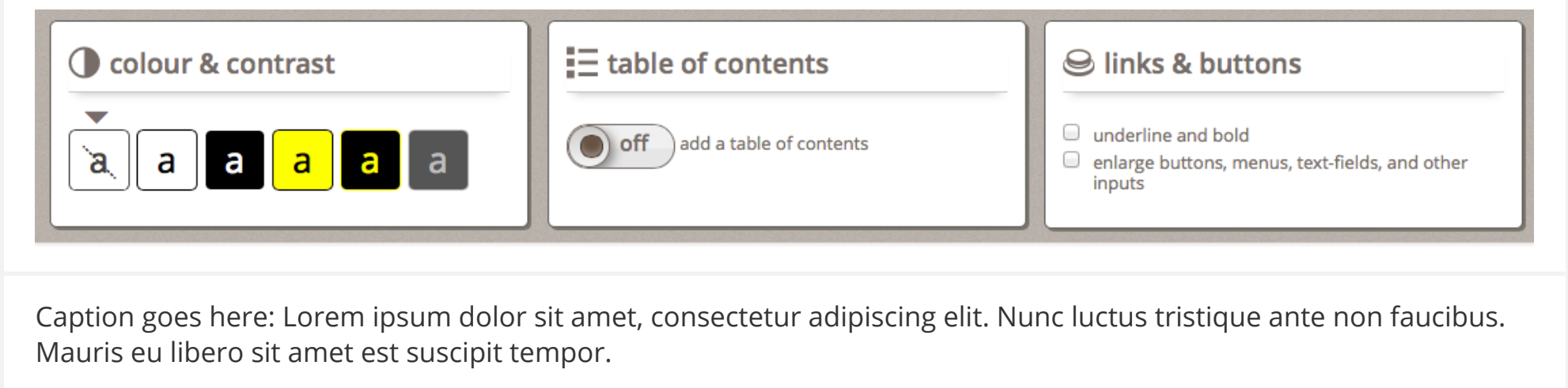
Examples with code

```
fluid.enhance.check({
  check1: "my.checking.function1",
  check2: "my.checking.function2",
  ...
});
```

The function `fluid.enhance.check()` will execute the specified functions and store the results in the static environment using the associated key (e.g. `check1`). The presence of the tags in the static environment can be used in the context argument to `fluid.demands()`.

Decorator Type	Field Name	Field Type	Field Description	Example
Decorator or Type	func	String	jQuery function to be invoked	decorators: [{ type: "jQuery", func: "click", args: function() {\$(this).hide();} }]
	args	Array of Object	Arguments to the jQuery function	

Tables and larger images can fill all available width if required.



Caption goes here: Lorem ipsum dolor sit amet, consectetur adipiscing elit. Nunc luctus tristique ante non faucibus. Mauris eu libero sit amet est suscipit tempor.