Views
ActionView is written in embedded Ruby with HTML and is responsible for compiling the response.

ActionView with Rails

app/views holds template files for the views associated with each controller.
generate scaffold article

Templates

.erb extension if embedded ruby and html both are used.
.builder extension if the Builder::XmlMarkup library is used.

.html.erb for ERB template system.

<% %> and <%= %> tags are used in ERB template.

1. For <%%> the code does not return anything. Useful for conditions, loops and blocks.
2. <%= %> are used when output is required.

```
<h1> Names: </h1>
<% @people.each do |person| %>
 Name: <%= person.name %> <br>
<% end %>
```

Builder

Helpful for generating XML content. Xml object is made available as well.

```
xml.em(“emphasized”)
xml.a(“Link”, “href” => “https://rubyonrails.org”)
```

Jbuilder

JBuilder (gem) is included in the default Rails Gemfile. Similar to JSON.
Can be added with gem ‘jbuilder’

Jbuilder object named json is made available to templates with .jbuilder extension too.

```
json.name(“Alex”)
json.email(“alex@example.com”)
```

Template Caching

When a template is altered, Rails checks modification time and recompiles if necessary.

Partials
Pieces of code are extracted and can be reused through templates. Are separated inn files.

Rendering partials can be done using the render method within the view.
<%= render “menu” %>

This renders _menu.html.erb (leading underscore tells that its different from a regular view).

<%= render “application/menu” %>
calls the partial from partial from app/views/application/_menu.html.erb.

Using Partials to Simplify Views

Concept: move the details out of a view to grasp things more clearly.

```
<%= render "application/ad_banner" %>

<h1>Products</h1>
<p>Here are a few of our fine products:</p>
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
<%= render "application/footer" %>
```

Content can be shared amongst pages as well (footer and add banner).

Render with locals Option

The locals: option includes keys as a partition-local variable.

<%= render partial: "products/product", locals: { product: @product } %>
If a passed variable is not part of this option, Template::Error is thrown.

Local_assigns

Keys within the locals: option are available through the local_assigns method.
<% local_assigns[:product] # => "#<Product:0x0000000109ec5d10>" %>

local_assigns is a Hash, it's compatible with Ruby 3.1's pattern matching assignment

local_assigns => { product:, **options }
product # => "#<Product:0x0000000109ec5d10>"

options # => {}

Pattern matching assignment supports variable rename

local_assigns returns Hash. Variables can be conditionally read and then fall back to a default value if key isnt a part of the locals:options.

```
<% local_assigns.fetch(:related_products, []).each do |related_product| %>
  <%# ... %>
<% end %>
```

Render without partial and locals Options

Before: <%= render partial: "product", locals: { product: @product } %>
After: <%= render "product", product: @product %>

The as and object Options

ActionView::Partials::PartialRenderer has an object in local variable with the same name as the template.

<%= render partial: "product" %>

Within the _product partial -> @product is received in the local variable product instead of having to write:
<%= render partial: "product", locals: { product: @product } %>

The object option can directly specify what object to be rendered into the partial.
<%= render partial: "product", object: @item %>

Using ‘as’ option a different name for the local variable can be used.
<%= render partial: "product", object: @item, as: "item" %>

Rendering Collection

A template iterates over a collection and renders a sub-template for each element.

To render all products:

```
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

Can be written as: <%= render partial: "product", collection: @products %>

When a partial is called with a collection, the individual instances have access to the member of the collection being rendered.
E.g in partial is _product, can refer to product to get the collection member

can use a shorthand syntax for rendering collections.
<%= render @products %>

Rails determines the name of the partial to use by looking at the model name e.g. product

Spacer Templates

Second partial can be rendered between instances of the main partials.
This can be done by :spacer_template option.

<%= render partial: @products, spacer_template: "product_ruler" %>

Rails renders _ product_ruler partial without any data passed to it between each _product partials

Strict Locals

For defining what locals a template accepts use ‘locals’ magic comment

<%# locals: (message:) -%>
<%= message %>

Disable locals: <%# locals: () %>

Layouts

Render a common view template.
For example: a different layout (different menu or navbar etc) for different users like end user, sales side etc.

Partial Layouts

Partials can have their own layouts applied to them (different from those applied to a controller).

Article.create(body: 'Partial Layouts are cool!')

articles/show.html.erb
<%= render partial: 'article', layout: 'box', locals: { article: @article } %>

Box wraps the _article partial in a div:

```
<div class='box'>
  <%= yield %>
</div>
```

The partial layout now has access to the article variable through render. (_ prefix necessary).

articles/show.html.erb does the same thing here:

```
<% render(layout: 'box', locals: { article: @article }) do %>
  <div>
        <p><%= article.body %></p>
  </div>
<% end %>
```

View Paths

View path by default -> app/views
To add other locations and give precedence by the prepend_view_path and append_view_path methods.

Prepend View Path

When putting views inside another dir for subdomains:

```
prepend_view_path "app/views/#{request.subdomain}"
```

Append View Path

```
append_view_path "app/views/direct"
```

Helpers

Ror gives helper methods to use with action view, e.g:

Formatting dates, strings and numbers
Creating HTML links to images, videos, stylesheets
Sanitizing content
Creating forms
Localizing content

Localized Views

Can render different templates depending upon the current locale.

Can set a local using: I18n.locale = :de
This renders app/views/articles/show.de.html.erb.
If there is no locale, the default is rendered: app/views/articles/show.html.erb

Creating special views like app/views/articles/show.expert.html.erb that would just be displayed to expert users

```
before_action :set_expert_locale

def set_expert_locale
  I18n.locale = :expert if current_user.expert?
end
```